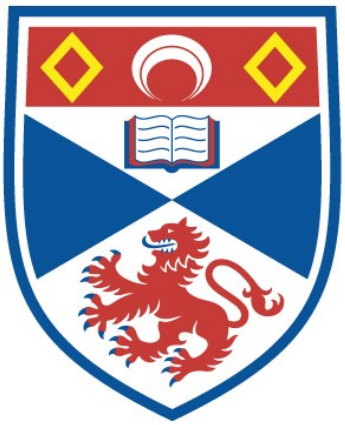
ON THE DEVELOPMENT OF ALGOL

Ronald Morrison

A Thesis Submitted for the Degree of PhD
at the
University of St Andrews



1979

ABSTRACT

The thesis outlines the major problems in the design of high level programming languages. The complexity of these languages has caused the user problems in intellectual manageability. Part of this complexity is caused by lack of generality which also causes loss of power. The maxim of power through simplicity, simplicity through generality is established. To achieve this simplicity a number of ground rules, the principle of abstraction, the principle of correspondence and the principle of data type completeness are discussed and used to form a methodology for programming language design. The methodology is then put into practice and the language S-algol is designed as the first member of a family of languages.

The second part of the thesis describes the implementation of the S-algol language. In particular a simple and effective method of compiler construction based on the technique of recursive descent is developed. The method uses a hierarchy of abstractions which are implemented as layers to define the compiler. The simplicity and success of the technique depends on the structuring of the layers and the choice of abstractions. The compiler is itself written in S-algol.

An abstract machine to support the S-algol language is then proposed and implemented. This machine, the S-code machine, has two stacks and a heap with a garbage collector and a unique method of procedure entry and exit. A detailed description of the S-code machine for the PDP11 computer is given in the Appendices.

The thesis then describes the measurement tools used to aid the implementor and the user. The results of improvements in efficiency when these tools are used on the compiler itself are discussed.

Finally, the research is evaluated and a discussion of how it may be extended is given.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

ON THE DEVELOPMENT OF ALGOL

by

Ronald Morrison

A thesis submitted for the degree of Doctor of Philosophy

Department of Computational Science

University of St.Andrews

St.Andrews

December 1979

Declaration

I declare that this thesis has been composed by myself and that the work that it describes has been done by myself. The work has not been submitted in any previous application for a higher degree. The research has been performed since my admission as a research student under Ordinance General No 12 on 1st. October 1971 for the degree of Doctor of Philosophy.

Ronald Morrison

I hereby declare that the conditions of the Ordinance and Regulations for the degree of Doctor of Philosophy (Ph.d.) at the University of St.Andrews have been fulfilled by the candidate, Ronald Morrison.

Professor A.J.Cole

Acknowledgements

A number of people must be thanked for their help with the S-algol project. Firstly, Professor Jack Cole, my friend and supervisor, whose enthusiasm and encouragement were responsible for the development of the system. Peter Bailey must be thanked for his help in testing the system and in proof reading, as well as his enormous willingness to help when difficulties arose. Paul Maritz, who is currently implementing S-algol on the Zilog Z80, pointed out a discrepancy in the application of the principle of correspondence to the language. Mike Livesey must also be thanked for our discussions of the semantic principles on which the language is based, during our daily runs. David Turner, now in Canterbury, with whom I started on language design deserves mention for his contribution. Whether he likes it or not his influence is present in S-algol.

These people have contributed to the language design as have the useful comments of Hamish Gunn, Tony Davie, Michael Weatherill and Ian Sommerville at Strathclyde University. My special thanks to Tony for his SPELL command and to the person who wrote the UNIX roff program with which this thesis was prepared.

Finally, my thanks to Ann, for David and for the proof reading of this thesis.

Contents

Appendices

I The S-algol Language Reference Manual

II Measurement Tools

III The Abstract Machine Code

IV S-code Generated by the S-algol Compiler

V The PDP11 S-code Operation codes

1. Introduction

An important area of interest to which Computational Science must address itself is that of programming languages. The situation is analogous to that in mathematics where notations are constantly being invented and improved. There has never been one mathematical notation or computer language which suits all tastes and needs, and it is unreasonable to expect that there ever will be. That does not mean that development of further languages is useless. Indeed better languages must be developed continuously in order to achieve greater clarity and levels of abstraction. Hopefully this can be done in a disciplined manner.

Landin [1] has pointed out that most languages have a lot in common and apart from syntax only a little in difference. He suggests that one way in which orderly development of programming languages can take place is by having a family of languages. This family of languages, in his case ISWIM (if you see what I mean), would have a common basic structure which could be altered to suit the application area. For example, most of the current programming languages have similar control structures. The difference between the languages is in the data structures that they allow. Therefore, a very simple minded example of a family of languages could be a fixed number of control structures with different data structures depending on the application. Indeed Landin goes on to say that Algol 60 would have avoided some needless criticism had it been defined as a family of languages rather than one all encompassing language. It is with defining and implementing a family base language that this work is concerned.

The notations that are used in the current general purpose programming languages are a balance between descriptive power and what the machines can effectively support. The descriptive power is borrowed from mathematics in general. However, mathematical notations tend to suffer from

the fact that they are expensive to implement on Von Neumann architectures. On the other hand, using a present day programming language involves ignoring most of the mathematical notions of correctness. Therefore the success of a general purpose language involves the question of how cheaply in machine terms can this descriptive power be bought. This is always at the back of the designers mind, and how much emphasis is placed on each aspect determines the flavour of the language.

The languages that will be considered here can generally be described as the algols. That is to say, low level languages such as assemblers and machine orientated languages such as PL360 [3] will not be discussed. Furthermore, the applicative languages such as the proposed ISWIM family, LISP [4] and the more modern sugarings of the lambda calculus like SASL [5] are also not considered. The discussion is restricted to the algols which can be roughly classified by the following rules.

a. Scope rules and block structure

Names may be introduced to define local quantities. The names are undefined outside the local environment. However, different environments may use the same name unambiguously to represent different objects.

b. Abstraction facility

The algols all have a powerful abstraction mechanism to allow the user to shorten and clarify programs. This is usually in the form of a procedure with parameters.

c. Compile time type checking

The types of all the expressions in the language can be checked by a static analysis of the program.

d. Infinite store

The programmer is relieved of the burden of storage allocation and is presented with a conceptually infinite store.

e. Selective store updating

In conjunction with (d) above, the user is allowed to selectively alter the store. This is usually implemented as an efficiency consideration on present day machines and generally takes the form of an assignment statement. It should be pointed out that it is this rule that gives rise to the concept of the store. Rule (d) may be implemented as an infinite capacity to create new objects as is the case with the applicative languages.

With any rough classification, some languages cut across the rules. How well a language fits the rules determines how much of an algol it is. Languages that could be considered as algols are Algol 60 [6], Algol W [7], Algol R [9,38], and perhaps Pascal [10], Algol 68 [11] and PL/1 [12].

Where have they gone wrong? Have they? Many criticisms of these languages have appeared, particularly for the more popular ones such as Pascal, Algol 68 and PL/1. They take the form of criticising particular aspects of the languages [14,15,16] or the overall design. Dijkstra [13] has this to say of PL/1

> "Using PL/1 must be like flying a plane with 7,000 buttons, switches and handles to manipulate in the cockpit. I absolutely fail to see how we can keep our growing programs firmly within our intellectual grip when by its sheer baroqueness the programming language - our basic tool mind you! - already escapes our intellectual control."

It is not part of this work to perform a character assassination on each of the algol programming languages but merely to report that something is wrong with them. A clue to this malaise is again given by Dijkstra [13]

> "Another lesson we should have learned from the recent past is that development of "richer" or "more powerful" programming languages was a mistake in the sense that these baroque monstrosities, these conglomerations of idiosyncrasies, are really unmanageable both mechanically and mentally."

The trend continues. The recently published American Department of Defence language ADA [30] has tried to solve every programming problem known to mankind. It is the PL/1 of our age.

The development of these languages is in itself a commentary on our attitude to research and development. One method of research is to extend the existing world of possibilities. In this, new languages with new or unusual ideas are developed and tested. In order that some new concept may be tried and fully understood it may be necessary, and is often useful, to be able to use it experimentally.

Another approach is to re-examine the existing world to obtain a

better understanding of it. This type of research is also necessary since development of new ideas without a firm understanding of the existing world is like building on sand and will suffer the same fate as the building if the sand shifts.

This work is concerned with a re-examination of the existing world to obtain that greater understanding and hopefully a stronger basis on which to develop a new language or family of languages. It is not the aim of the work to produce a mechanism for extending language but rather with producing a base language and a methodology for further development.

Algol 60 was the first of the algol like languages. The project was a tremendous success and has led to the proliferation of many derivatives and extensions. However it would be unreasonable to expect the designers of the very first algol to be correct in every detail. Dijkstra hints at the problem

> "With a device as powerful as BNF, the report on the Algorithmic language Algol 60 should have been much shorter."

He means that Algol 60 is in some way too big. This, of course, is with the benefit of hindsight but it is with that knowledge that new languages are developed. Since then, very few languages have come anywhere near the brevity of definition of Algol 60. Indeed the language definitions of Algol 68, ADA and PL/1 are so perverse and long winded that they tax one person's understanding too far. The phenomenon of the "one liner" or the "algol lawyer" is a direct consequence of this.

Another manifestation of long winded languages is that of "dialects". Each implementor prunes the language to his own taste and restricts the possibility of machine independence. From time to time the

ridiculous situation arises that a standard for the language is introduced as has recently happened with Pascal [19]. This leads to the obvious question of what happened to the original language?

The method of attack that will be taken in this research will be that of simplicity. If a concept is not already simple then it should be re-evaluated. Algol 60 is in some respects already too complex for our purposes and will have to be re-examined. This will take some courage as it may mean disposing of the services of some favorite feature since it does not measure up to the yardstick of simplicity. On occasion some very useful concept may fit the general rules but fail the test of simplicity. In order to progress it may be necessary to leave it as a complex entity (that concept may simply be complex). At least it will have been analysed thoroughly by the new methodology of language design and presumably a better understanding of the concept will have been achieved.

How should this language be designed? Wirth [2] has pointed out that a language designer is faced with a bewildering variety of demands. He lists some of them as

The language should be easy to learn

It must be safe from misinterpretation and misuse

It must be extensible without change to existing facilities

The notation must be convenient with widely used standards

The definition must be machine independent

The language must make efficient use of the computer

The compiler must be fast and compact

The definition must be self contained and complete

The time and cost of development must be minimal

There are others not mentioned here. The list can be extended and usually is by the implementor to include his particular preferences. The

emphasis that is placed on each aspect of the design usually determines the language that is arrived at. It can be clearly seen that although the above aims are desirable they do not provide a methodology for programming language design since they are in Wirth's own words a bewildering variety.

In this work there will be only one design aim for the language. This aim is an economy of concept where the language will have a small number of general rules with no features or special cases. Occam's Razor will be applied wherever possible to preserve the language simplicity.

From the above it can be seen that Wirth lays the emphasis in language design on the implementation. In fact he goes as far as to argue that "language design is compiler construction". While it is true that the popularity of some fairly appalling languages such as BASIC [17] and APL [18] has been due to their good support systems, this view seems a little extreme. The problem is that it confuses the design aims of the language with the design aims of the implementation.

Having said that, it should be pointed out that the implementation is still very important. The most successful of the modern languages are those that reflect best what our present day machines can support efficiently. Already concessions have been made to efficiency, by the inclusion of the assignment statement in the languages under consideration, and it will be the reason for introducing a restricted form of pointer. However, efficiency is a fickle bedfellow. The ground rules may change drastically with the invention of a new piece of hardware. A decision on the form of a language from efficiency considerations should only be undertaken with extreme care.

Wirth is therefore not totally wrong and it is only his emphasis that is disagreed with. Language design cannot be completely divorced from the implementation and it is part of this work to investigate the software

engineering aspects of language implementation. In particular, a simple method of compiler construction must be found and a general method of implementation developed. Measurement tools and programmer aids must also be investigated. The commercial success of the language will depend on the implementation process. However, the language itself may be evaluated separately.

The description of the work falls naturally into four parts. Chapters 2 and 3 discuss the design of a particular language, S-algol, and the design methodology. Chapter 4 develops a simple method of compiler construction and chapters 5 and 6 consider the system implementation. The tools for measuring the system are discussed in chapter 7 and chapter 8 reviews the research and assesses it.

2. Language Design Using Semantic Principles

Language design is probably the most emotive area in Computational Science. Nearly everyone uses programming languages and most people have something to say about their design. This fact is reflected in the literature. Most of the interest is centred on criticising existing language features, only a little in developing new languages and almost none on a methodology for language design. Certainly there is plenty of general advice like "the language must be easy to learn" but this is so obvious that it is hardly worthwhile saying.

A. van Wijngaarden [20] with his notion of a generalised algol may be used as a starting point.

> "In order that a language be powerful and elegant it should not contain many concepts."

He argues that languages are too complex and that complexity is due, in part at least, to being too restrictive. Power through simplicity, simplicity through generality is the message. This leads the language designer to formulate the fundamental concepts behind the language and to generalise these ideas wherever possible. Some of the consequences of following this course are surprising and can lead to complications.

In both Euler [8] and Algol 68, the procedure is regarded as a first class data object. It was realised that some power could be gained by passing procedures as parameters to other procedures. This, of course, is very like assignment of procedures so the generalisation is to have proper procedure variables and allow the procedure values to be passed around freely. Quite soon the idea of function producing functions [21] appears and before we know it a fundamental decision on the implementation has been made. It turns out that languages with function producing functions, proper

algol scope rules and assignment cannot be supported by a stack machine architecture [22]. However, the language is fundamentally "richer" or more powerful in that it can support higher order functions.

At this stage the designer has to make a decision. The concept generalisation can either be ignored on some other constraint, for example implementation efficiency, or else he has to live with the consequences. This is the only manner in which simplicity will be achieved. Unfortunately both Euler and Algol 68 introduce further rules on the restriction of free variables to allow the facility and maintain ease of implementation. This is also true of the generalised pointer concept of Algol 68 and is perhaps a clue to why it is such a big language.

Why then would we wish to restrict a generalisation? One case where it is advisable is in compile time type checking. The following example is due to Henderson [23] and is written in an algol like language.

```
begin
     procedure P( Q,a )
     begin
          a[ 1 ] := 0 ;
          Q( a )
     end ;

     procedure R( x )
     begin
          x := x + 1
     end ;

     array b[ 1::10 ]

     P( R,b )

end.
```

This program attempts to add 1 to a one dimensional array. If the language specified its parameter types in full, this nonsense could be detected at compile time. It is perhaps surprising that Algol 60, Algol W and Pascal would allow this program. Complete generality can lead to the

situation where little can be deduced from the program on static analysis. This is bad since most of our concepts of notations depend on a supportive static representation.

However, the overall design aim of power through simplicity, simplicity through generality should be the guiding light. On some occasions other aims may restrict the generality. Wirth [2] argues that such an occasion is the conflict between security and flexibility exemplified by the above example.

The general rules for the design of programming languages by semantic principles will now be discussed. All of the principles follow the guiding star proposed above.

2.1 The Principle of Correspondence

This first rule is based on work done by Strachey [25] and is more clearly reviewed by Tennent [24]. However, the problem was first stated by Landin [1].

> "In almost every language a user can coin names, obeying certain rules about the contexts in which the name is used and their relation to the textual segments that introduce, define, declare or otherwise constrain its use. These rules vary considerably from one language to another, and frequently even within a single language there may be different conventions for different classes of names with near analogies that come irritatingly close to being exact."

Landin points out that all the rules governing names in a language should be designed together in order to avoid irregularities in

the manner in which the names may be used. For example, the scope rules should be the same everywhere. In most algols names may only be introduced in declarations and as procedure parameters. Therefore for each type of declaration in the language there should be a corresponding parametric declaration. Indeed parameters should be regarded as locally defined objects. The principle of correspondence goes a little further by taking account of all possible parameter passing modes.

This rule obeys the simplicity yardstick and Tennent performs a comparison on Pascal parametric and declarative objects. Unfortunately Pascal is found to be lacking under this type of analysis since most of the declarative constructs have no parametric equivalent.

2.2 The Principle of Abstraction

This rule has the same sources as the previous one. Abstraction is a process of extracting the general structure to allow the inessential details to be ignored. This facility is well known to mathematicians and programmers since it is the only tool they have for handling complexity.

The technique when applied to language design is to define all the semantically meaningful syntactic categories in the language and allow an abstraction over them. The most familiar form of abstraction is the function which is an abstraction over expressions. The principle of abstraction is more difficult to apply than the principle of correspondence since it is more difficult to identify the semantically meaningful constructs than it is to identify declarations. Also it is often difficult to visualise the use of an abstraction once it is made.

However, the principle of abstraction is kept as a general rule and a justification must be made if it is broken.

2.3 The Principle of Data Type Completeness

This principle, while not explicitly stated by Strachey, is one he was well aware of. The rule states that all data types must have the same "civil rights" in the language and that rules for using data types should be complete with no gaps. This means that any operation such as assignment or passing the data object as a parameter has an equivalent form for all data types. The rule does not require that all operations be defined on all data types but rather that special cases of general rules are non-existent.

Examples of lack of completeness can be seen in Algol W where arrays are not allowed as fields of records and in Pascal where only some objects are allowed as elements of sets. This principle will lead to simplicity since it avoids the complexity of special cases.

2.4 The Conceptual Store

Another aspect which will effect the semantic model of the language is that of the conceptual store. Since this discussion is about the algols it is already committed to a store. However, it is useful to look at how this arose as it will give a clearer understanding of why it is there.

In general, mathematics has no concept of a store. There are only expressions which are characterised by having a value. The most useful property of only having a world of expressions is that of referential transparency described by Quine [26]. This states that in order to evaluate an expression with sub-expressions all we require to know about the sub-expressions is their values. Anything else, such as the order of evaluation is irrelevant. This property is an old friend in mathematics and is often used unconsciously as Strachey points out

sin(6) sin(1 + 5) and sin(30 / 5)

are all expected to have the same value. It means that the sub-expressions may be evaluated in any order. The applicative languages make use of general mathematical notation and have no underlying concept of a store. Donahue [27] emphasises that the semantic model for these types of languages are much simpler and Hoare [28] has proposed a model for data structures in such an environment.

How then does the concept of the store arise? Of course, it comes from the design of computer hardware with high speed stores. To allow the power of these machines to be fully utilised it is necessary to make as best use of the high speed store as possible. It is therefore an efficiency consideration.

The assignment statement is introduced to allow the store to be reused. There are two places where this will improve the efficiency of a program. Firstly in loop control to alter the controlling value and secondly in pointers to allow selective updating of data structures. Just as the goto statement is considered harmful in our programming languages and is replaced by higher order constructs, it may be possible to do the same for assignment. Hehner [29] proposes a model for an algol like language without the concept of a store. Whether it can be implemented efficiently is an area of further research. Since the store may be altered during the dynamic evaluation of the program it also introduces the concept of sequencing. Hehner's model also suffers from this and it could be that removal of the store in his case does not really gain the required results.

Finally the store introduces commands or statements in the algol sense. Quite often in our programming languages the statement is the natural unit of expression. However, statements should be regarded as mere syntactic sugarings to help us write down more complex expressions. Indeed

it is helpful to view the statement as an expression with a void value. It would be useful if the language encouraged this in some manner.

Having discussed a number of rules for language design we may proceed thus :

a. Data Types

Decide which data types, both simple and compound, are required by the language. Define the operations on the data types and check that the principle of data type completeness holds.

b. The Store

Introduce the store and the manner in which it may be used. This includes defining the statements in the language and such issues as protection.

c. Abstraction

Define the semantically meaningful syntactic categories and invent abstractions for each.

d. Declaration and Parameters

Invent the declarations and the parametric objects together. The issues here are store protection and parameter passing modes having a one to one correspondence with declarative modes.

e. Input and Output

Introduce the I/O model to the system.

f. Iterate

> Finally, re-evaluate the language and correct or justify any idiosyncrasies in the design. How often the rules can be broken is for the designer's own conscience. The rules were introduced to help design simpler and more intellectually manageable languages and should only be ignored with great care.

Once this has been achieved and the language has reached a fixed point, a concrete syntax may be invented. It is perhaps desirable that different groups of workers have a different concrete syntax depending on their taste. In any case the syntax should enhance the clarity of programs not hinder it.

3. S-algol Design

The S-algol language is designed on the principles outlined in the previous chapter. A discussion of the main concepts of the language is now given. As with most other languages there is a tension between the design aims and a practical implementation. Whenever one of the principles has been violated a complete discussion of the issues involved is given. However, in the main, adherence to the three tenets has led to a very simple language based on the fact that there are fewer rules and exceptions. An assessment of the language is given in chapter 8.

The description of the main issues in the language follows the method of development. However, since the language exists, it has a concrete syntax for at least one implementation. Where it is felt necessary this concrete syntax has been used for clarity in the description. The full syntax is given in Appendix I with the language reference manual.

The discussion begins with the design of the data types.

3.1 The Universe of Discourse

The data types in S-algol are defined by the following rules.

a. The scalar data types are int, real, bool, file and string.

Type int represents the values in the set of all integers.In practice the integers may be restricted to the arithmetic limit of the particular hardware. The same is true of type real which represents the values in the set of all reals. The operations on int and reals are addition, subtraction, multiplication, division and remainder for ints. The integers and the reals have the relational operators less than, less

than or equal,greater than, greater than or equal,equal to and not equal to defined on them but the result is of type bool. Finally type int may be coerced to real if necessary.

Type bool represents the values in the set { true,false }. The operations on bools are equal to, not equal to, and, inclusive or and not. The 'and' and 'or' are non-strict.

Type file represents the set of file descriptors that are available in a particular implementation. The only operations on files are equal to and not equal to.

Finally, type string represents the set of all possible collections of characters in the character set. At first this seems to be a compound data object and is regarded as such by some. However, it is unnecessarily complex to introduce a type char and a compound type string for all types. The concept is too close to a vector for comfort and is further confused by the difference between vectors of characters and strings of characters. By including strings and not characters as a scalar data type this confusion is avoided and a simple method of allowing the basic operations on strings is available. These operations are concatenation, substring selection, length and all the relational operations. This approach is much more pleasant than having, as in Pascal, arrays of characters.

b. For any data type T, *T is the data type of a vector with components of type T

> The operations on vectors are indexing, upper and lower bound, equal to and not equal to. Note that the bounds of the vector are not part of its type and are determined dynamically. Also multi-dimensional arrays can be implemented as vectors of vectors.

c. The data type pntr comprises of a structure with any number of fields and any data type in each field.

> Each user defined structure class has a fixed number of fields of fixed type. The class of a structure, like the bounds of a vector is not part of the pntr type but is again determined dynamically. The operations on structures are field selection, equal to, not equal to and a test that a pntr is a given class.

To ensure the principle of data type completeness is followed the world of data objects is given as the closure of rule (a). under the recursive application of rules (b) and (c). This, of course, gives an infinite number of data types. Anywhere in the language where a data type is referred to, care must be taken to ensure that all data types are acceptable. This is not applicable to operations on specific data types. In this manner all data types will have the same civil rights.

Notice that unlike Algol 68, equality is defined on all of the data types. A full explanation of this is given in the next section.

3.2 The S-algol Conceptual Store

The store concept, in the algols, arises from the practicality of efficient implementations of programs on Von Neumann machines. One of the areas in which efficiency will be improved is in the implementation of large data structures. In S-algol, the vectors and structures have full civil rights and may be assigned. From an efficiency point of view it is unwise to copy these objects on assignment and a slightly different view of vectors and structures must be taken for an efficient implementation.

Objects of type *T and pntr are regarded as pointers to vectors and structures respectively. The value of a vector or a structure is defined as the pointer. Therefore on assignment or similar operations only the pointer i.e the value is copied.

The definition of equality on all data types can now be regarded as a comparison of their values. However, it must always be remembered that the value of a compound data object is the pointer. This gives further proof that the store complicates the semantic model.

Furthermore, the problem of the general pointer of L-value as Strachey would call it has been raised. Let us quickly kill it by using as evidence against, the objections raised by Hoare [28] and the disastrous effects it has had on Algol 68 such as the dangling reference. It is bad enough having a restricted pointer for efficiency's sake.

The semantic model of the store is one with L-values and R-values for each data type. That is, the values in all data types may be assigned to a location. In general L-values cannot be passed around only assigned to. However, vectors and structures contain L-values which themselves contain the R-values of the elements.

One further concept to be discussed in relation to the store is protection and in particular constancy.

3.3 Constancy

The concept of a variable is one which is well established in the current algols. It takes the form of a name which may alter its value during the execution of the program. The variable is implemented by a location which contains its value. When a variable is updated its R-value is altered. Constants, by comparison, are not often found as a separate concept, although the majority of variables would be better implemented as constants since they never alter their value. A constant is an object whose value is invariant. It may be initialised but never altered in its lifetime. Like a variable, a constant has an L-value and an R-value, but any attempt to update it will produce an error.According to Strachey [31]

> "Constancy is an attribute of the L-value,and is moreover an invariant property. Thus when we create a new L-value, ... , we must decide whether it is variable or constant."

The manifest constants of BCPL [32] and Pascal do not meet this definition of constancy. They are compile time objects and are not implemented as protected locations. Furthermore, the issue of constancy extends beyond the world of scalar data objects. As the definition points out, anywhere an L-value is introduced it must have the option of being constant or variable. Therefore, in addition to the scalar values being constant so also may structures, their fields, vectors and their elements be constant. A fuller discussion on the concept of constancy is given by Gunn and Morrison [33].

3.4 Control Structures

The introduction of the store forces consideration of the language control structures. Legard and Marcotty [36] have classified control structures and S-algol falls into their D′ category. Their conclusion is that the correct balance between power and security is given by this category. S-algol is an expression orientated language and it regards a statement as an expression of type void. The selection clauses are *if* ..... *then* ..... *else* which degenerates to *if* ..... *do* for the single pronged version, and a *case* clause. The *case* clause is a new type of case statement where the case selector is of any type and is matched with expressions of the same type to find the selected clause. The matching test is the equality test and the tests are performed in order so that the programmer can place the most likely one first. This is rather like the guarded commands of Dijkstra [57] with the non-determinism removed.

The rest of the control structures are fairly conventional. They are *while* ... *do* , *repeat* ... *while* and *repeat* ... *while* ... *do* which give loops with tests at the start, end and middle of loops. The *for* loop is similar to that proposed by Hoare [34]. The control identifier is constant and is redefined every time round the loop. The initial value, step and limit are restricted to integers and evaluated only once. Of course, there is no *goto* statement.

3.5 Abstractions

Tennent [24] has suggested that the method of applying the principle of abstraction is to identify the semantically meaningful syntactic categories in the language and allow an abstraction over them. This he does for Pascal and proposes some extensions to complete the

abstractions. However, he points out that it is not an easy matter to identify these categories in the first place. A table is given for S-algol

Syntactic category	Abstraction
typed clause	typed procedure
untyped clause	untyped procedure
declaration	module
sequencer	sequel

Procedures are familiar, the abstractions over declarations and sequencers are not. Tennent use the name module after Schuman [35] for a declaration abstraction. The problem with this is that it destroys the block structured scope rules of algol. His abstraction over sequencers is a sequel which is just another convoluted goto. Legard and Marcotty [36] and now Morrison are extremely critical of this type of branch. On the other hand Knuth [37] defends it.

The problem now is to identify the useful abstractions. The procedure is an old and trusted friend. The sequel looks dangerous and confusing. The module at first glance looks useful but it destroys the scope rules. In the final analysis S-algol rejected the module and the sequel. It is interesting to note that the language has only four semantically meaningful syntactic categories which perhaps highlights its simplicity.

3.6 Declarations and Parameters

Before discussing this, some support from Dijkstra [13] for the approach taken is given. First of all the Algol 60 parameter mechanism.

> ".... I am getting very doubtful about Algol 60's parameter passing mechanism ; it allows the programmer so much combinatorial freedom that its confident use requires a strong discipline from the programmer. Besides being expensive to implement it seems dangerous to use."

and later on in the same paper.

> "A number of rules have been discovered, violation of which will either seriously impair or totally destroy the intellectual manageability of the program ..... Examples are the inclusion of the goto statement and of procedures with more than one output parameter."

In the modern algols a plethora of parameter passing modes has evolved. It is usually one of the most difficult areas to understand in a language and is certainly complex to teach. By considering the declarative and parametric mechanisms together and applying, the principle of correspondence, it is hoped that a simpler and more elegant solution will be found.

The output parameter is the first to receive attention. Algol W calls this a result parameter. It seems strange indeed to have an object of this type at all. It is for most of the time a name without a value. As a

declaration it is usually given as a name to be bound to a type. The value is made by an assignment later. Disallowing this forces all declarations to initialise the object, which is a good idea since it completely eliminates one type of programming error and also improves efficiency since the implementation does not have to check for uninitialised values.

Value-Result parameters suffer from a similar nonsense. They are merely a variation on call by reference. Since, in general, L-values have been banned from being passed around, the parameter mode has no declarative analogy. Also languages without this mode will not suffer from the Fortran disease of overwriting literals.

Declaring a name and giving a value or calling a procedure sets up a correspondence between the formal parameter (name) and the actual parameter (value). It was argued above that names should be given values on declaration. This corresponds to call by value for parameters. Assignment to a formal parameter inside a procedure has no effect outside and the value will disappear on exit. With both declarations and parameters an L-value has been introduced and the syntax must allow for it being constant or variable. This does not effect the parameter passing mode it merely determines whether assignments to the location are allowed. Thus S-algol has only call by value with the procedure being able to return a value which may be scalar or compound.

It may be argued that call by reference is also present for vector and structure elements. This is because a decision has already been taken to have reference objects and causes no inconsistency between declarative and parametric modes. The value of a vector or a structure is its pointer and the reference idea is already present in assignment. Indeed it would be inconsistent to implement the parameter in any other manner.

Finally on declarations, names may be introduced to represent a

structure class and a procedure. To complete the principle of correspondence there must be a parametric equivalent. In order to preserve the strong typing in the language, a procedure passed as a parameter must specify its parameter types and the same is true for a structure class. This looks very like having structure classes and procedures as first class citizens. However, these objects were specifically excluded from the description of the data types in order to avoid the complication of implementing function producing functions [22] and modals [23] in Algol 68 parlance. They are merely here to complete the principle of correspondence. It may also be noted that passing a function as a parameter is the correct method of implementing call by name. The table below gives the S-algol modes.

Denotation	Declarative construct	Parametric Formal	Construct Actual
Initialised name with constant value	let I = E,for I = E....	cT I	E
Initialised name with variable value	let I := E	T I	E
structure	structure I(tlIl,....tnIn)	same	I
procedure	procedure I(tlIl,....tnIn)	(tl...tn)I	I

3.7 The Input Output Model

The I/O models for most high level languages tend to reflect the environment in which they were designed. Some attempts have been made to design and implement comprehensive I/O systems. Unfortunately where it has not been tied to particular hardware, as in the Algol 68 case, it has never been very successful. Nowhere else in the design of a programming language does the hardware intervene as much as it does in the I/O system. When a new I/O device becomes available the language must be able to make use of it. Of course, this situation is hopeless and perhaps the wisest approach to I/O is to allow the implementor to deal with it for a particular environment, as the Algol 60 designers proposed.

The S-algol solution is to propose an I/O model but to invite the implementor to alter the model in the spirit of the language whenever it it considered necessary. By designing an extremely simple I/O system it is hoped that together with the abstraction facilities in the language, it will be powerful enough to handle any environment. This is perhaps a forlorn hope.

The S-algol I/O system is based on files. A file is a sequence of characters or binary digits. Files may be created, deleted, and generally manipulated within a program. A file descriptor is a data type with full civil rights. The file system has functions which act on files to perform the I/O. Read and Write are two of these functions and they exist for data types int, real, bool and string in both character and binary form. The test for equality on two files, tests if they are the same file.

This very simple system is extremely powerful especially when combined with the S-algol abstraction facilities. However it is doubtful if it is sufficient to handle all the situations that may arise. Furthermore, the file system functions do not act on all data types which breaks the

principle of data type completeness. This is a strong indication that more work is required on this problem.

3.8 The Concrete Syntax

The final stage of language design is to propose a concrete syntax. Ideally different groups of workers could have a different syntax. However, there are many users who do not wish to design their own syntax and so the language must provide at least one possibility.

It seems very obvious to say that the syntax should be simple and easy to learn. That may be so but there is no doubt that some of the success of the language depends on the cosmetics. Also, a carefully chosen syntax can ease the problem of compilation. Wirth [2] has this to say.

> "Adhere to a syntactic structure that can be analysed by a simple technique such as recursive descent with one symbol look-ahead. This not only aids the compiler, but also the programmer, and is vital for the successful diagnosis of errors."

The design of the syntax is also a balance between brevity and clarity. An example of this design tension is illustrated by the different rules for introducing names in S-algol.

When an object is declared, its name, its value, its type and whether it is a constant or variable is required by the compiler. The problem is how concise can this be made without being obscure. For declarations S-algol has

```
let I := E
```

where I is the identifier and E is any valid expression.

e.g.

```
let a := 2
```

tells the compiler that a variable, a, of type int and initial value 2 has been introduced. The compiler does not need to be told the type, it can deduce it from the declaration enabling it to be brief. For constants = is used instead of :=.

e.g.

```
let a = 13.2 * 2.15
```

introduces a real constant.

It is possible for the compiler to deduce the types of all the data objects without the program ever having to explicitly mention them. But is it wise?

Take for example a procedure heading. The compiler can deduce from the calls of the procedure, the parameter types and whether they are consistent. However, it cannot deduce whether the formal parameter is constant or variable. Furthermore, the procedure should be understandable without reference to the calls. Therefore it is sensible to force the user to specify the object type and whether or not it is constant. Finally, if the procedure returns a value it is aesthetically pleasing to have the object type specified in the procedure heading. It should be emphasised that this should not be confused with the principle of correspondence as it is merely fitting a convenient syntax around the semantic model.

The declarations in the algols is an area which always causes difficulty. In S-algol there is no goto clause and therefore the programmer cannot jump round a declaration. By insisting that everything is declared before it is used except in the case of mutually recursive procedures, the rules on the position of a declaration can be relaxed. In S-algol,

declarations may be freely mixed with statements. The scope of a name is from immediately after the declaration to the end of the sequence. This allows names to be introduced as locally as possible. Again it should be noted that this is merely a syntactic extension.

The full language description is given in Appendix I and a solution to Wirth's string problem is now given as a sample of the flavour of the syntax. The problem is to write an interactive program to read a string length and to find all the strings of that length of three characters with no adjacent repeated substrings.

```
let s := "" ; let n := 0 ; let not.done := true

procedure add.a.letter ; { s := s ++ "a" ; n := n + 1 }

procedure alter
case s( n|1 ) of
"a"     : s := s( 1|n-1 ) ++ "b"
"b"     : s := s( 1|n-1 ) ++ "c"
default : { n := n - 1 ; if n = 0 then not.done := false
                                  else alter }

procedure acceptable( -> bool )
begin
     let ok := true ; let p := 1
     while ok and p <= n div 2 do
     begin
          ok := s( n - p + 1|p ) ~= s( n - 2 * p + 1|p )
          p := p + 1
     end
     ok
end

!                  Main Program

write "Input string length "
let lnth = readi ; let count := 0
while not.done and n <= lnth do
begin
     if n = lnth then alter else add.a.letter
     while not.done and ~acceptable do alter
     if n = lnth do { write "An acceptable string is ",s,"'n"
                      count := count + 1 }
end
write "Total number of strings of length ",lnth : 3,
      "is ",count : 4,"'n"
?
```

Notice that by having lexical rules that allow a semi-colon to be omitted if it coincides with a newline, most of the semi-colons in the algols can be avoided.

4. Compiler Construction

Wirth [2] has hinted that judicious selection of the concrete syntax of a programming language will ease the difficulty of compilation. The syntax of S-algol has been carefully chosen in order that it may be compiled by the technique of recursive descent. It is not part of this work to develop the theory of LL parsing, an excellent account of which is given by Griffiths [40], but rather to investigate how easily it can be implemented. However, as background a short discussion of recursive descent is now given.

4.1 Recursive Descent

Traditionally compilers have been constructed along the lines of the first portable compiler BCPL [41]. Algol W [42] and Whetstone Algol 60 [43] are also examples. The compiler is constructed in three stages.

a. lexical analysis

b. syntax analysis

c. code generation

The stages may run as coroutines but more commonly act as three separate passes on the source code. This is shown diagrammatically in figure 4.1.

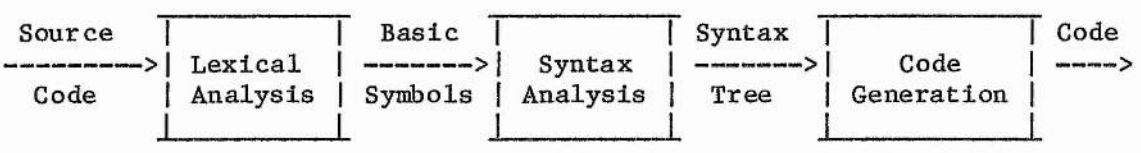


Figure 4.1

The lexical analysis converts the source code into basic symbols from which the syntax analyser will generate a syntax tree. Finally, the code generator converts the syntax tree to machine code.

The advantage of a recursive descent compiler is that it eliminates the need for an explicit syntax tree. The "tree" is contained in the recursive evaluation of the compiler program itself. For every syntactic construct there is a procedure in the compiler which will analyse it. When a procedure to analyse a particular language construct is called, the data area (stack frame) it uses will implicitly define the branch of the tree that is of interest. This eliminates the code to set up and interrogate the syntax tree and therefore makes the compiler smaller. It is a short step to do the same for the lexical analysis and make the compiler one pass.

This, of course, makes the compiler faster but the real significance is that it makes the total code for the compiler smaller and easier to write. This results in better written, more correct, more portable and easier to understand compilers.

The structuring technique for such a compiler was first outlined by Ammann [44] although he was not the first to write a recursive descent compiler. The compiler is refined in layers rather than as separate passes. Each layer is added to the code rather like a coat of paint, until the compiler is complete. Whereas the general method of construction is borrowed from Ammann, the particular method differs somewhat and is an extension of that developed by Turner and Morrison [45,38]. In particular, the chosen layers are different and the coding technique is entirely new. Instead of locally defining the compiler functions, they are defined globally to allow any other function to access them. This was done for clarity and understanding. Also the abstractions used to code the compiler,

which is its big advance as far as software engineering is concerned, owes nothing to Ammann's work.

The layers to define the compiler are.

a. Write a pure syntax analyser

b. Write a lexical analyser

c. Add the context free error diagnosis and recovery

d. Add the type checking and type handler

e. Add the environment handler and scope checker

f. Add the context sensitive error reporting

g. Add the data address calculation

h. Write the code generation

The technique is now described by using the if clause as an illustration.

The BNF of the if clause is

if < clause > then < clause > else < clause >

with type rule

if { BOOL } then { T } else { T } => { T]

The compiler is written in S-algol itself and that will be used in the description of the layering.

4.2 Syntax Analysis

Recursive descent compiling constricts the syntax to being LL. Where it is not LL(1) , the parser may cheat and look ahead. However, it is generally a good idea to keep as close to LL(1) as possible. Some systems [58] have been developed to help mould a grammar into LL(1).

The technique of compilation works by defining a procedure for each non terminal in the syntax. This procedure will process the non terminal by calling procedures recursively to handle the other non terminals in its definition and by parsing terminal symbols. The syntax analysis is therefore performed as the recursion is created with the implicit tree in the stack of the called procedures.

Although the lexical analyser is not yet written, a number of functions are defined in order that the terminal symbols in the syntax may be parsed. These are

```
procedure have( string S -> bool )

! if S is the symbol in the input stream then advance
it and return true. Otherwise do not advance the input
stream and return false.

procedure mustbe( string S )

! if S is the symbol in the input stream then advance
it. Otherwise report an error.

procedure syntax( string S )

! write a syntax error report. S is the required
symbol.

procedure next.sy

! place the next basic symbol in the global string
variable symb.
```

These functions have been carefully chosen as they are the level of abstraction at which the syntax analyser will view the lexical analyser. If these are not chosen carefully, the structure of the compiler may be preserved but the coding clarity will not.

To parse the if clause, a procedure of a similar name is called. The syntax analysis will have recognised the if symbol and the procedure must progress from there.It is

```
procedure if.clause
begin
     next.sy
     clause
     mustbe( "then" )
     clause
     mustbe( "else" )
     clause
end
```

which follows the BNF

if < clause > then < clause > else < clause >

closely.

There are only a few departures from the BNF, the most notable being the parsing of expressions. In order that the recursion does not get too deep, the expressions are parsed by a mixture of recursive descent for expressions of different precedence and operator precedence (looping) for expressions of the same precedence. Also the assignment clause is parsed by

< expression > ::= < clause >

and the excessive generosity corrected later by the type matcher. This technique for expressions is borrowed from the BCPL compiler [41].

The structure of the compiler is based on the syntax analysis. By refining the syntax procedures using the abstractions of each stage, the various functions are added to the compiler. Of course, at any stage the compiler can be run as a program and the implementor may feel that this is useful for such a large program. Before the discussion turns to refinement, the lexical analysis which has already been used is described.

4.3 Lexical Analysis

The lexical analyser is one of the machine dependent sections of the compiler. It need not be written at this stage but since it has already been used in the abstract perhaps it is better out of the way.

The lexical analyser forms characters in the input stream into basic symbols in the language. That is, it must recognise names, reserved words, literals of type int, real, bool and string with all their lexical conventions, symbols such as ~= and all single character symbols. Also it must elide tabs, spaces, comments and compiler directives as well as organising the printing of the program.

Since it is the aim of a one pass compiler to remove the intermediate forms from the compiler, a basic symbol is represented by the string itself. The variable symb is used to hold the current symbol. If the basic symbol is an identifier or a literal, symb has the value "identifier" or "literal" respectively, with the actual value in the variable the.name or the.literal depending on the type. One concession to efficiency is that for every reserved word and some other symbols there is only one copy of the string and a string constant with that value. Thus the compiler has defined for the else symbol

```
let else.sy = "else"
```

and else.sy is used instead of "else" to save space.

The lexical analysis is viewed from the rest of the compiler through the procedures have and next.sy. Next.sy processes the input stream and forms a basic symbol which it places in symb. Have, places the next symbol in symb if the present one is equal to the parameter and returns the value true. Otherwise it merely returns the value false.

4.4 Syntax Error Diagnosis and Recovery

This technique is based on one invented by Turner [47]. He points out that the diagnosis of, and the recovery from, context free errors can be achieved at very low cost. The strategy is that when no alternative symbol in the input stream will do for correct syntax analysis, the compiler can detect any errors. The procedure mustbe is used to detect that the required symbol is the same as the one in the input stream. If the symbols are the same, the next symbol is placed in symb for parsing.

When an error is detected a suitable message is printed by the procedure syntax. Recovery from an error is performed by allowing the compiler to continue thinking that the symbol was correct. When mustbe is called again and a match is made, recovery is complete. No further error messages for context free errors are printed in between. If a match is not made on the second call of mustbe the input stream is scanned until it is found.

The method is very low cost and will catch and recover from all first order errors. E.g. a missing bracket. Turner points out that the advantage of the scheme is that it can be tuned wherever necessary and can be made more sophisticated depending on the syntax in use.

The problem with error recovery is that an incorrect program is being altered to what the compiler thinks is the nearest correct one in order to detect further errors. Before embarking on a complex error recovery scheme, such as Pascal employs or as outlined by Backhouse [46], the designer has to be convinced that it is worthwhile and that the strategy can be explained to the user.

4.5 Type Checking

Superimposed, as a refinement, on the syntax analysis are the type checking abstractions. The basic types are discussed in Chapter 3 but must be extended to allow for complete type checking. Type void is the type of a clause with no value i.e. a statement, and there are also names of type procedure, structure class and structure field in the language.

For good error reports, the types are represented in the compiler by structures. Each scalar type has a structure whose field is the string name of the type.

e.g. The type int is represented by

```
let int.sy = "int"
let INT = scalar( int.sy )
```

Every time type int is used, it is represented by the constant INT for readability , efficiency and good error reporting.

Each procedure in the syntax analysis now becomes a function which returns the type of the object compiled. In the same manner that the syntax analysis is performed , as the recursion progresses, the type checking is done on the return journey. Therefore, each procedure must check the type of every syntactic element it causes to be parsed at a lower level.

The abstract form of the type checker which is used by the syntax analysis procedures takes the form of two procedures called match and eq. Match compares the two structures it is given which represent types and issues an error message if they are not compatible. The procedure eq performs the comparison on the two types and returns a boolean result. Since there are an infinite number of types in the language the procedure must be recursive. The types of non scalar types are represented by

building up layers of structures to identify the compound object. This recursive data structure may be analysed by eq.

The if clause has the type rule

if { BOOL } then { T } else { T } => { T }

The code therefore becomes

```
procedure if.clause( -> pntr )
begin
     next.sy
     match( BOOL,clause )
     mustbe( then.sy )
     let t = clause
     mustbe( else.sy )
     match( t,clause )
     t
end
```

Notice that if t is void, the if statement has been compiled otherwise the if expression.

The context sensitive errors that occur at this time are found and reported by the procedure match. When two types do not match an error message is printed and care has to be taken to avoid a plethora of messages thereafter. Type ANY is used by the compiler for cases where the type is not known and the type checker will match ANY with any other types in subsequent attempts at matching.

It is also convenient to perform the checking for constancy at the same time as the type checking. By making a small change to the language, that of vectors with constant elements and vectors with variable elements being assignment incompatible, the checking for constancy can be performed at compile time. The compiler must check for constancy on assignment.

Finally, the type checking must handle coercions from type int to type real where necessary. This will complicate the code for the if clause when the code generation is added.

4.6 Scope Checking and Environment Handling

Each level of scope of the program being compiled is represented in the compiler by a binary tree. Each element on the tree represents a user defined name. For each name the following information is eventually required

a. The name itself

b. The object's address pair

c. The type and constancy attribute

d. A pointer to the left and right neighbours

The levels of scope are chained together in a linked list. Entering a new level of scope merely adds a link to the list and leaving a level gets rid of the current tree as it is no longer required.

The abstractions used at this level are implemented by the functions declare and lookup. The procedure declare takes a name and its type and enters it in the tree. The function lookup will check that a name is in scope and if it is it will return its type. Context sensitive errors may be detected here and error reports may have to be issued.

For good error reporting some names may have to be checked for duplication before they are declared. For example the simple declaration

let I := E

The name I does not come into scope until after the declaration. Also the type is not known until the expression E is compiled. However, to obtain a good and early error report the name must be checked for duplication on finding the := symbol. The function check.name is used for this purpose.

4.7 Data Address Calculation

A decision about the abstract machine has now to be taken. The technique of recursive descent lends itself to generating code for a reverse polish machine such as the beta machine described by Randell and Russell [43].

Each data item on the stack is represented by an address pair. The abstract machine is described in the next chapter and it is sufficient here to mention that the address pair consists of the lexicographic level of the item and its offset from the level base. When a stack item is declared this address pair is calculated.

Items such as structure classes also have an address. The structure class has an address called a trademark which is an index into the structure table. The structure table has one entry for each structure class which contains information on the layout of the structure fields. Thus the trademark uniquely identifies the structure class. The trademark is kept on the stack and the address of the structure class is the address of the trademark. Fields on the other hand require both the structure class address and the offset of the field from the base of the structure. The structure table which describes the structure classes is output by the compiler at the end of the compilation.

Care must be taken during the address calculation that the difference between compile time and run time is not confused. An example of this is given in section 4.8

4.8 Code Generation

The final layer of refinement in the development of the compiler is that of code generation. The compiler produces code for the S-algol stack machine which is described in Chapter 5 with the code for each syntactic construct described in Appendix IV. The code generation is written in such a manner that code for any machine could be produced. For each abstract machine instruction a procedure is defined to simulate the execution of the instruction and to generate code for it.

The simulated evaluation of the code is necessary to keep track of such things as the stack pointer. Since declarations can occur mixed with clauses it is important to remember the stack position of the declaration. During the generation of the code it is also necessary to keep track of the code addresses for jumps. This requires slight modification depending on the target machine. The method used in the S-code machine is now described.

When code is generated it is held in a large vector in the compiler until the segment being compiled is complete. At this point, since there are no goto's in the language, all the jumps will be resolved relative to the segment base. The segment is then written out and replaced by its closure, see later, on the stack. The address of the segment is relative to the program base. Inside the segment the jumps are only generated by high level constructs. This means that forward and backward references may be easily identified. For backward references , the position of the reference is remembered in the procedure compiling the syntactic construct and is merely generated with the jump. Forward references are more difficult and require a backward linked list to the reference to be set up in the code whenever it is used. This list of references is resolved when the point of the reference is reached. The technique avoids the

complication of label tables etc.

To isolate the code generation as far as possible, the work is done inside the code generation procedures with calls from the syntactic analysis. The code generated by the compiler for the if clause is

if E0 then E1 else E2 => E0 jumpf(1) E1 jump(m) setlab(1) E2 setlab(m)

The if clause now becomes

```
procedure if.clause( -> pntr )
begin
     next.sy
     match( BOOL,clause )
     let l = jumpf( newlab )
     mustbe( then.sy )
     let t = clause
     dec.stack( t )
     let m = fjump( newlab )
     mustbe( else.sy )
     setlab( l )
     match( t,clause )
     setlab( m )
     t
end
```

Procedure jumpf simulates the execution of the abstract machine instruction jump if false. It also removes the boolean from the top of the stack. The jump is always forward so the parameter is the chain so far, in this case a new label, and the result is the position in the code of this reference. This will be resolved by the procedure setlab just after the else symbol has been parsed. A similar situation occurs with the label m except that the jump is unconditional and the stack is not involved. An example between dynamic evaluation and static analysis can be observed here. After the then symbol has been parsed, the result of the clause must be removed from the stack during static analysis. This is performed by the procedure dec.stack. Of course, dec.stack does not generate any code as only one of the two paths will be followed dynamically.

4.9 Conclusion

The technique of compiling by recursive descent and building the compiler by stepwise refinement was successful. It resulted in a fast, small and easy to understand compiler. This is borne out by the code to compile the while clause

BNF

```
while < clause > do < clause >
```

and type rule

```
while { BOOL } do { VOID } => { VOID }
```

and code

```
while E0 do E1 => E0 setlab(1) jumpf(m) E1 jump(1) setlab(m)
```

gives

```
procedure while.clause( -> pntr )
begin
     next.sy
     let l = cp
     match( VOID,clause )
     let m = jumpf( newlab )
     mustbe( do.sy )
     match( VOID,clause )
     bjump( l )
     setlab( m )
     VOID
end
```

With the example of the if clause as the only previous explanation of the compiler techniques , it is easy to understand and have confidence in the correctness of the code for the while clause. The total compiler is about 1500 lines of S-algol laid out rather like these examples. This compares very favourably with most other compilers.

The structuring and refinement of the compiler is not quite as

smooth as Ammann suggests. In particular the design of certain layers have to be done in conjunction. For example, it is unwise to proceed with the syntax analysis without designing the abstract lexical analysis and a similar situation occurs with the type matching and the environment handler. Therefore, the layers are not as separate as may be desirable.

Furthermore, the choice of abstractions at the coding level, of which Ammann says nothing, contributes more to the conciseness or verbosity of the compiler than does the structuring. In other words, the successful compiler depends on the correct level of abstraction being identified and refined by the implementor and not merely the fact that the technique is recursive descent. It just so happens that recursive descent lends itself well to this type of treatment.

5. The S-algol Abstract Machine

The architecture of any abstract machine is determined by the power of the language it has to support. The recursive nature of the block structured languages, like the algols, lends itself to implementation by a stack such as described by Hauck and Dent [48]. Most of the implementations of the algols are based on variations of the beta machine of Randell and Russell [43]. However, it should be noted that the beta machine is not sufficient to support languages like Algol W, Pascal, Algol 68 and S-algol which although predominantely stack based, require a second area of dynamically allocated store. This area is known as the heap. Many variations of the beta machine exist e.g. Pascal P-code [49] and wherever possible the S-algol machine has drawn on their experience. The overall design tenet was to design a simple machine to support the stack and heap nature of S-algol. The resultant machine is the S-code machine. The description of the S-code machine falls into the three categories of stack organisation, heap organisation and the instruction set.

5.1 The Stack

The S-algol compiler produces S-code which is a form of reverse polish instruction code. The S-code is ideally executed on a stack machine. The stack is used to facilitate block and procedure entry and exit, to provide space for programmer named objects and to provide space for expression evaluation. Expression evaluation is always performed on the reusable space at the top of the stack and since the technique is well known little more will be said of it here.

On block or procedure entry or exit, information is placed on or removed from the stack. This information which contains a Mark Stack Control Word (MSCW), space for local objects and parameters and working

space for expression evaluation, is sometimes known as a stack frame. Since the length of each stack frame may be different, they must be linked together to allow correct exit from the block or procedure. Therefore, the MSCW contains a dynamic link which points to the base of the previously activated stack frame. Thus, the dynamic links form a chain, known as the dynamic chain, of the currently activated blocks or procedures.

By its very nature the stack records the dynamic evaluation of a program. Some method is required to reflect the static nature i.e. the scope rules, since not all of the stack frames available on the stack need be in scope. The MSCW contains a second pointer, known as the static link, which points to the stack frame of the immediate static outer block or procedure. These static links form the static chain which can be used to find the stack frame base of any block or procedure that is in scope.

The position of a stack frame, for a block or a procedure, on the stack may vary depending on the dynamic evaluation of the program. Therefore, the compiler cannot calculate the absolute address of stack items other than those of the most outer block. However, if dynamic vectors are disallowed as stack items (it will be shown later that S-algol vectors cannot be implemented on the stack anyway), the address of a stack item relative to its stack frame base may be calculated at compile time.

For each item on the stack the compiler produces an address pair

< ll,dd > where

ll is the lexicographic level and

dd is the displacement from the stack frame base

This address pair is used at run time to calculate the absolute address of the stack item. Note that if the stack frame base of the item was to be found by chaining down the static chain, the clever compiler [49]

would calculate, instead of ll, the difference in the current lexicographic level and that of the required item as this is the number of times to link down the static chain.

The next refinement is to use fast registers to form a display [48]. The display duplicates the values in the static chain and thus the absolute address of any item on the stack is

display(ll) + dd

Finally, Wichmann [50] has shown that stack frames are only required for procedures as blocks can be considered as part of the procedure and their stack item addresses calculated relative to the procedure stack frame base. This technique is called procedure level addressing and extends with a slight modification to languages with block expressions.

5.2 The S-algol Stack

S-algol was designed to be used to write programs in the paradigm of structured programming [51]. By studying the programs written in such a manner , two observations which are relevant to this discussion can be made. Firstly, structured programs tend to consist of a large number of small procedures which are called many times. Secondly, the objects referred to in these procedures tend to be local or outer block globals. It is therefore reasonable to design an abstract machine which takes advantage of this to obtain an efficient implementation.

The display method of implementing the stack has two major drawbacks. Updating the display may be complex. In general, the number of operations required to update the display on procedure exit is

p - q + 1 if q <= p and 0 otherwise

where p is the ll of the calling procedure

and q is th ll of the called procedure

In other words, if the environment changes drastically, the overhead in updating the display is increased. This situation occurs on returning from a procedure declared at a lower level than the calling one and with procedures passed as parameters. Some compilers e.g. Algol W [42] panic because of this and simply dump the whole display on to the stack on procedure entry and restore it on exit.

The other main difficulty of the display scheme is that it relies on there being enough spare registers on the target machine to reflect the depth of static nesting in any program. In most cases this usually leads to some arbitrary restriction on the depth of nesting. While this may be good enough for hand written programs it is almost never satisfactory for automatically produced programs.

The S-algol abstract machine requires two registers, called stack base, SB, and stack front, SF, which point to the global and local stack frame bases respectively. Only one register, SF, is absolutely necessary since the base of the stack may be fixed. On procedure entry SF is made to point to the new stack frame base and on exit it is restored from the dynamic link.

Thus, since the compiler issues separate instructions for locals and outer block globals, they can be quickly accessed by using the SF and SB registers. Other free variables are accessed via the static chain. The method is as fast as the display technique for accessing locals and outer block globals and does not suffer from any artificial limit on the static depth.

A simple and efficient method of procedure entry and exit is now possible. However , the technique must be capable of dealing with procedures passed as parameters. Consider the program to calculate

$$\int_1^2 \int_3^4 e^{xy} dxdy$$

```
procedure integral( ( real -> real )f ; real a,b -> real )
( b - a ) * f( ( a + b ) / 2.0 )

procedure g( real z -> real )
begin
     procedure ep( real y -> real ) ; exp( z * y )

     integral( ep,2,4 )
end

write integral( g,1,2 ) ?
```

The problem arises on the call of the procedure integral which has ep as a parameter. Procedure ep uses the non local y and therefore when ep is invoked as procedure f an environment change is required to obtain the correct results. The difficulty is to devise a general mechanism which will deal with this case. The S-algol solution is as follows.

Since a procedure can be passed as a parameter and is subject to the same scope rules as other items, it suggests that it could be implemented as a stack item. The above problem is one of identifying the correct static environment. When a procedure is declared two items are placed on the stack as an initialisation. These are the procedure address and its static link and are collectively called the procedure closure. Each procedure forms a segment of code which is loaded separately. The only evidence of one segment having been part of another is the instruction to

load the closure. The static link may be calculated at this point as it is merely the current stack frame base. This can simply be copied and need not be recalculated on each procedure call.

A procedure call consists of an instruction to load the closure, code to evaluate the parameters and an instruction to call the procedure. The MSCW contains

a. The procedure address

b. The static link

c. The dynamic link

d. The return address

The first two items form the procedure closure and the second two are calculated just before entry. The closure uniquely represents the procedure on the stack. A procedure declared inside another will appear at run time as a closure on the stack. To pass a procedure as a parameter requires the closure to be copied. Thus, the procedure parameter will look like a locally declared procedure when it is called. For the closure to represent the procedure uniquely, it is not sufficient to hold the procedure address ; the static link is also required. The procedure closure carries its own static link and therefore the environment problems are solved.

The technique also works for forward declared procedures by leaving space on the stack at the point of declaration and filling the closure in at the point of actual declaration.

Procedure exit is extremely simple. Resetting the SF pointer is all that is required. No updating of the static chain is required as it is uncovered to the position on the point of entry automatically. Of course, stack retraction must take place on procedure and block exit and

care has to be taken if they return values.

Whether this technique is more efficient than the display method depends on

a. The number of spare registers on the target machine

b. The number of non global free variable accesses

c. The number of procedure calls

d. The number of procedures passed as parameters

and is discussed further by Morrison [52].

5.3 The Heap

The design of the world of data objects in S-algol, makes it impossible for the language to be implemented on a stack only system. In particular, strings, vectors and structures require a second area of dynamically allocated store known as the heap.

In S-algol, vectors are first class data objects. They enjoy the same civil rights as any other data object including assignment, being fields of other vectors or structures and being passed to procedures. The copying of vectors is not generally used on efficiency grounds. Indeed, on a stack system with variable size vectors, as in S-algol, it is impossible to copy the vectors since the space required to hold the vector on the stack cannot be predicted. Therefore, a vector is represented by a pointer on the stack with the elements on the heap. The value of the vector is the pointer and assigning the vector means copying the pointer. Since pointers are of uniform stack size the problem is overcome.

The S-algol structures suffer from the same problems as the vectors. Indeed the language is designed to make vectors and structures behave in the same manner. Structures may be of any class and therefore

size. For exactly the same reasons as with vectors , structures are implemented as pointers on the stack which point to the structure fields on the heap.

Since the programmer may alter the fields of a structure or the elements of a vector , it is necessary to know that vectors and structures are implemented as pointers. For example, a vector passed to a procedure which alters one of the elements has that element altered forever. Strings, on the other hand, are pure data objects and may not be altered internally by the programmer. However, strings also have the same size problems as vectors and structures. They are therefore implemented on the heap with a pointer to them on the stack. The pointer in this case is not seen by the programmer.

The efficiency of this approach to strings, vectors and structures is not immediately obvious. The problem of copying large data structures has been avoided, in general, by the introduction of the pointer. The pointer is only apparent to the user with vectors and structures since they may be altered internally. With strings the pointer is hidden and is used only for efficient implementation.

The abstract machine data structures on the heap must also contain some housekeeping information to allow them to be used correctly. They are listed for each item

a. Vectors must carry their bounds for run time bound checking and an indication if the elements are pointers for garbage collection purposes.

b. Structures must carry their trademark for run time structure class checking and some means of specifying which fields are pointers for structure creation and garbage collection.

c. Strings must carry their size for index checking.

How this is implemented efficiently on a given target machine is a problem for the ingenuity of the implementor. The PDP11 solution is given in Chapter 6.

5.4 Heap Organisation

The algol family of languages present the user with a conceptually infinite store. The stack simulates this by reusing the store allocated to blocks no longer in use. The design philosophy of S-algol does not wish to alter this. Since the difference between stack and heap objects is hidden from the user, the heap as well as the stack must be reused. Of course, the pretence breaks down when the store is finally exhausted and may not be reused. The technique of garbage collection is used with the heap.

Space is allocated on the heap until there is no more available. At this point it is possible to have space allocated that is no longer in use. Consider the program section

```
for i = 1 to 10000 do
begin
     let A = @1 of cstring[ "a","b","c","d" ]
     .
     .
     .
     .
     .
end
```

After executing the loop, 10,000 versions of A will be on the

heap. However, if none of them has been assigned then none of them will be in use. The same applies to structures and strings.

At any point in the execution of an S-algol program, the space on the heap may be

a. Allocated and in use

b. Allocated and not in use

c. Free

When the free space is exhausted it is the job of the garbage collector to free the space that is allocated and not in use. This is performed in two stages by marking and collecting.

In the marking phase, the pointers on the stack are used to identify some heap items that are in use. The search is recursive since heap items may point to other heap items. Thus all the space that is in use will be marked. Once the heap has been marked all the unmarked space is freed with the possibility of coalescing the free areas into larger blocks or compacting the used space to one end of the heap.

The main difficulty in this is to identify the pointers on the stack at any point in the execution of a program. Some pointers are trivial to find as they are at a fixed location relative to the stack frame base. However, the position of some other pointers on the stack is dependent on the dynamic flow of the program. Consider the example

```
structure vecs( *int V1 )

let A = vecs( @1 of int[ 1,2,3 ] )
```

If a garbage collection strikes between the creation of the vector and the structure , the pointer to the vector will be at an arbitrary position on the stack and therefore difficult to identify. A

solution could be to chain all the pointers on the stack together but this will slow down the use of items with pointers. Of course, a tagged data architecture machine has no difficulty with this problem. S-algol proposes a new, simple and extremely obvious solution.

The S-algol system has a separate stack for all pointers. The compiler can always predict the type of an item and therefore on which stack it will live. The marking algorithm has no difficulty in finding the initial pointers since they are all on the pointer stack. Thus a potentially awkward situation is overcome.

The drawbacks to the two stack solution are as follows.

a. A third area of dynamically allocated store must be found in the total address space. This is often no problem at all.

b. The second stack must be administered like the first to map the static and dynamic flow of the program.

The solution to the second problem is to allocate two registers, pointer stack front, PSF, and pointer stack base, PSB, to point at the current pointer stack frame base and the global pointer stack frame base. The main stack already maps the dynamic and static flow of the program. By including in the MSCW a pointer to the equivalent pointer stack frame base, all the intermediate pointer stack frames may be found by linking down the main stack static chain and then using the pointer to the pointer stack frame.

The mark stack control word now contains

a. The procedure address

b. The static link

c. The dynamic link

d. The pointer stack link

e. The return address

This, of course, complicates procedure entry and exit. However, the problem of identifying these anonymous pointers is such a nasty one that the price paid is felt to be small.

5.5 The Abstract Machine Code

The S-algol abstract machine uses four storage areas

a. The instruction code area

b. The main stack

c. The pointer stack

d. The heap

It also has seven special purpose registers

a. Stack front	SF
b. Stack base	SB
c. Pointer stack front	PSF
d. Pointer stack base	PSB
e. Stack top	SP
f. Pointer stack top	PSP
g. Code pointer	CP

The S-code generated for each syntactic construct is given in

Appendix IV and the S-code instructions are described in detail in Appendix III. A general discussion of the more unusual item of the S-code design is given here.

5.6 The Stack Instructions

The S-code machine has the usual battery of stack instructions. There are arithmetic instructions, like plus, which perform the operation on the top of the stack and leave the result there. The relational operations such as less than etc. operate on ints, reals and strings and leave a boolean result on the top of the stack. There are instructions to load literal values on to the stack and instructions to load other stack items on to the top of the stack prior to being used. This last group requires three forms since the stack objects may be local, global or intermediate depending on their scope. The type of the operand is used to indicate which stack is to be used.

There are a number of miscellaneous instructions such as erase the top of the stack, exchange the top two elements and assign the top element to the address given in the second top. A new stack instruction is retract which is used on block exit.

No code is required in procedure level addressing for block entry. However, on block exit the stack top registers may have to be changed to get rid of any locals on the stack. Furthermore, if the block returns a value this must be copied to the new stack top.

Finally, to allow a general coercion from type int to real a float instruction which will float the top or second top stack element is required.

Consider the expression

```
1 + 2.0
```

This will generate the code

```
ll.sint( 1 ) ll.real( 2.0 ) float.op( 2 ) plus.op
```

5.7 The Heap Instructions

The heap instructions fall into two categories

a. Those which create heap objects and therefore may cause garbage collection

b. Those which use the heap items

The string operations which create strings on the heap are code which creates a string of length 1, concat.op which concatenates two strings forming a new one, substr.op which selects characters from a string to form another and finally read.string. Creating structures is more complicated.

e.g.

```
structure abc( int a ; cstring b ; *real c )

let A = abc( 1,"ron",@1 of real[ 1,2,3 ] )
```

The code generated for the structure creation is

```
load the trademark from the stack
evaluate the expressions
form.structure( n )
```

The trademark is loaded on to the top of the stack and is n words

from the stack frame base. The expressions are evaluated on the appropriate stack depending on their type. The trademark must carry some indication of which fields are pointers in order to take the fields off the correct stack when filling in the fields. This information is also required in the marking phase of the garbage collector. An indication of how this might be done is given in Chapter 6.

There are two forms of vector creation. The first

```
@1 of int[ 2,3,45 ]
```

generates the code

```
ll.sint( 1 ) ll.sint( 2 )
ll.sint( 3 )
ll.sint( 45 )
make.vector( int,n )
```

The expression types are int and therefore on the main stack. n gives the position of the lower bound on the main stack relative to the stack frame base. It is a simple matter to calculate the size of the vector, create it on the heap and fill in the elements. The second form of vector creation is given by

```
vector 1::10,2::11,..... of "abc"
```

which generates

```
evaluate the bound pairs on the stack
evaluate "abc"
iliffe.op( t,n )
```

t is the type of the expression to be the value of each of the

elements of the last bound. n is the number of bound pairs. This instruction is not very easy to implement as the creation of the vector entails the recursive creation of the constituent vectors.

There are a number of instructions to access the heap items, the most important being the subscripting operations which have to check the vector bounds, the structure class or the string size. The run time structure class checking of S-algol is performed by these instructions.

5.8 Flow of Control Instructions

These instructions are necessary to map the rich set of high level language constructs in S-algol which alter the program flow of control. The first pair of these instructions is used to implement the non-strict version of and and or

E1 or E2 => E1 jumptt(1) E2 1:

Since true or anything gives true and false or anything gives anything, jumptt branches if the top stack element is true and merely removes it otherwise.

Another S-code instruction in this style is the instruction used to implement the case clause, cjump

```
case E0 of                              E0
E11,E12,.....E1n : E10        E11 cjump(11) E12 cjump(11).....E1n cjump(11)
                              jump(M1) 11 : E0 jump(xit)
E21,E22,.....E2n : E20      M1:E21 ..........
  .
  .
  .
  .
  .
  .
  .
  .
  .
default : Ek+1 0             Mk:Ek+1 0
                            xit:
```

The cjump instruction takes the top two stack elements, the stack dependent on the type, and if they are equal removes them both and jumps. Otherwise it removes only the top element.

An unconditional jump and a jump if the top stack element is false is sufficient to implement the if and loop clauses.

if E1 then E2 else => E1 jumpf(1) E2 jump(m) 1: E3 m:

and

while E1 do E2 => 1: E1 jumpf(m) E2 jump(1) m:

Another variation is required to allow the coercion from int to real in all cases. consider the example

if E1 then 1 else 2.0

It is not until the 2.0 is compiled that the compiler realises that 1 should 1.0. The following code sequence is applicable to the general case.

E1 jumpf(1) 11.int(1) jump(m) 1:11.real(2.0) jump(n) m:float.op(1) n:

The for clause is controlled by two instructions, one to perform the test at the beginning of the loop and one to perform the step at the end. The code generated for the for clause is

for i = E1 to E2 by E3 do E4

E1 E2 E3 1:fortest(m) E4 forstep(1) m:

The control constant, limit and increment are at the top of the main stack. The fortest instruction decides if the loop is finished and

jumps if it is. This is used in conjunction with the forstep instruction at the end of the loop which adds the increment to the control constant and jumps back to the beginning of the loop. Notice that the limit , increment and initial value are only calculated once before the start of the loop.

Finally, under flow of control instructions the code sequences to perform procedure entry and exit are examined. The code to call a procedure is

```
mst.load
evaluate the parameters
apply.op( m,n )
```

The mst.load instruction, of which there are three forms depending on the scope of the procedure, loads the closure on to the top of the stack and leaves space for the dynamic link, the static link and the return address. After the parameter expressions have been evaluated on the stack, apply.op is used to call the procedure. The numbers m and n give the positions of the new values of SF and PSF relative to SP and PSP respectively.

Apply operates like this

a. Fill in the dynamic link with the value of SF
b. Calculate new SF = SP - m and PSF = PSP - n
c. Fill in the pointer stack link with PSF
d. Fill in the return address
e. Move the contents of the stack location pointed at by SF to CP. This will perform the branch.

The code for the procedure itself is surrounded by the two instructions enter and return. Enter has two parameters which are the

maximum amount of stack space, on each stack, that the procedure may require. The enter instruction checks that the space is available. On some machine architectures e.g. Multics this is not necessary. The return instruction is more complex and works like this.

a. Move the result of the procedure, if any, at SP or PSP to the new stack top at SF or PSF and set SP and PSP to these values.

b. Move the dynamic link to SF

c. Move the pointer stack link of the uncovered frame to PSF

d. Move the return address to CP

5.9 Input and Output

The S-code machine supports binary and character I/O streams. These are implemented as files in the language and allow the user to read or write the appropriate data types. Read functions are defined to read ints, reals, bools and strings and some others to manipulate the stream. The write functions give the equivalent power for output and contain some additional formatting capability. The objects in character form may be written out in specified field widths with more powerful functions for the real.

The abstract machine does not define how the file should be implemented as it is felt that this may constrain the best environments e.q. UNIX to the low level service of some systems. It is therefore left to the implementor to organise this.

5.10 Conclusions

It can be readily seen that the S-code machine retains the spirit of the beta machine for its stack environments. The display mechanism is however gone. It was necessary to add a heap to the machine to implement vectors, structures and strings. The problem of identifying the pointers on the stack at garbage collection time lead to the implementation of a second stack for the pointers only. This complicates the abstract machine but it is felt that it solves the original problem so well that the complexity is a good investment.

The implementation of S-algol on a machine with two stacks and a heap requires that the abstract machine code will be different from other abstract machine codes. However, a lot of the instructions are common to most reverse polish machines and it is only the architecture of the S-machine that makes the instructions different. There are also some new instructions which allow the more esoteric S-algol constructs to be implemented at a fairly high level.

It was one of the design aims of S-algol, that the user should not know or not need to know, the difference between stack and heap objects. Under the rule of orthogonal design all the data types have the same rights. It is therefore necessary to reuse the heap just as the stack is reused. This introduced the garbage collector which in turn led to the invention of the two stacks. Thus, the two stacks and a heap architecture is arrived at.

6. S-algol Implementation

This section describes the implementation of S-algol on the PDP11 computer running under the UNIX operating system [53]. The main constraints on this system are that the PDP11 is a 16 bit word machine with a user address space maximum of 64K bytes. This immediately gives rise to space problems. Otherwise the UNIX environment is very friendly and is especially suited to the implementation of S-algol files.

The space constraint of the PDP11 leads to a severe restriction on the size of program that can be run. A compact form of code is required and an interpretive system is the solution. The compiler produces a form of S-code which is then interpreted. Of course, since the compiler is written in S-algol, it is itself interpreted. In Chapter 8, this decision is justified by the subsequent measurement of the system. Speed has been traded for space.

The S-code machine is laid out in store thus

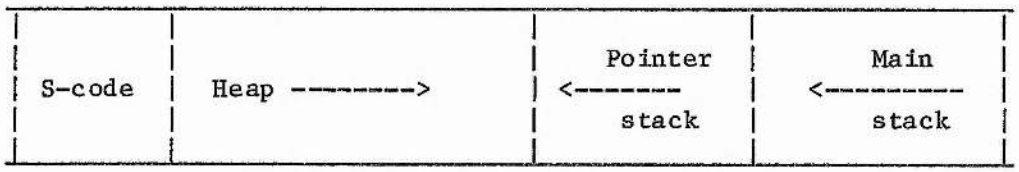


Figure 6.1

The sizes of the various components are roughly 4K bytes main stack and 2K pointer stack with an 8K interpreter. The S-code size depends on the program being run and the heap uses the rest of the 64K bytes.

The PDP11 registers are used thus

r0	general	r1	general
r2	code pointer CP	r3	pointer stack front PSF
r4	stack front SF	r5	pointer stack top PSP
r6	stack top SP	r7	PDP11 program counter

Note that there are not enough PDP11 registers to accommodate SB and PSB which have to be implemented as store locations.

6.1 Instruction Set Design

The instruction set codings were chosen in the first instance for simplicity. This is based on the belief that the only method of performing optimisation is to measure the system first to find out where to optimise. The first solution to the problem should involve the designer's intuitive ideas of efficiency but this should not be of overriding importance at least until some measurement statistics are available. It is therefore the aim to design a simple system which can be easily measured.

The PDP11 is a 16 bit word computer with the capability of addressing half words or bytes. It was decided, in the first instance, to design the instruction set so that one word would hold a complete S-code instruction. It was realised that this might not be the most efficient method since using a byte orientated system with long and short instruction forms may produce a better solution to the design aim of compact code. Only measurement of the results would provide the answer to this.

The constraints on such a design are that in order to fit both the operation and the address into 16 bits, the address must have a fixed maximum size. This, along with the knowledge of the work done by Wilner [54] on bit addressable machines led to three formats for the instruction

set. These are

4 bit instructions

bit 15	1	to identify these
bits 12-14		operation code
bits 0-11		code address

This format of instruction is used for the jump instructions. There are only eight codes. The jump address is restricted to 4096 which is not very large. Since S-algol programs cannot jump out of the procedure segment then the jump address can be relative to the segment base. Thus the restriction of 4096 bytes of interpretive code in each segment is quite large.

6 bit operations

bits 14-15	01	to identify these
bits 10-13		operation code
bits 0-9		stack address

This format is used for the instructions which involve a stack address. These instructions such as local, global, plocal etc. are restricted to a ten bit address. That is, the maximum number of stack elements per stack frame is 512.

9 bit operations

bits 14-15	00	to identify these
bits 7-13		operation code
bits 0-6		type

These instructions usually have the type of the instruction operand encoded in the bottom seven bits. Of course, some instructions do

not fit any of these formats and usually have the first word in a format with a second word of information following immediately.

This type of organisation requires some involvement by the compiler. It was decided to simplify affairs by having no loader. The compiler would produce the S-code which would be directly executable by the interpreter. This is always difficult for a one pass compiler compiling a blocked structured language. Consider the example

```
procedure a
begin
      .
      .
      .
      procedure b
      begin
            .
            .
            .
            .
      end
      .
      .
end
```

The problems are twofold.

a. The code will be produced with the procedures mixed up and

b. Forward references are difficult to resolve.

The first problem is that the procedures are not separate code segments. This has been traditionally solved [43] by planting a branch round the inner procedure so that it will not be executed at the point of declaration. That solution aggravates the problem of the maximum code displacement from the segment base being 4096 bytes.

The second problem is that a forward reference cannot be resolved once the code has been written out. The solution is the same for both problems.

A vector is used in the compiler to hold the code produced.

Forward references can therefore be back patched in this vector. At the end of the procedure declaration , the code for the procedure can be output and the instruction to load its closure left in its place. Thus the procedures are independent segments of code.

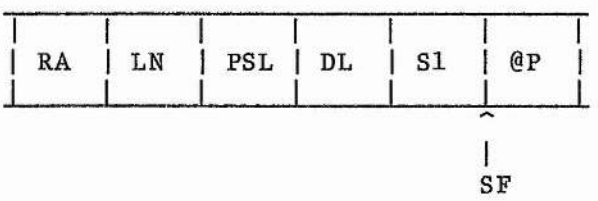
The effectiveness of the instruction set design is discussed in Chapter 7 but as a first guess it seems to have been good. On a static analysis of the S-code for the compiler itself, the number of bits required to represent all the operation codes was 58,542 against a theoretical minimum [55] of 43,096.

A description of each instruction in the PDP11 S-code is given in Appendix IV and the instruction codings in Appendix V.

6.2 The Stacks

The PDP11 computer has instruction modes which allow the store to be manipulated as a stack. These addressing modes are autoincrement which fetches the operand and increments the register pointing at it, and autodecrement which decrements the register and uses it as the address of the operand. One consequence of this is that the PDP11 stacks run backwards from store high addresses to store low addresses. Stack items not at the top of the stack are usually addressed by a displacement. In the PDP11 implementation this displacement is subtracted from and not added to the stack frame base to find the correct item address.

The layout of the MSCW is now



The position of SF is convenient for procedure exit. The line number LN in the MSCW is the number of the line that was executing prior to the procedure being called. In the PDP11 system there is a location which contains the number of the line being executed. This location is updated by the newline instruction. On procedure entry and exit the location value must be stored in the MSCW and restored from it respectively. It is used for diagnostic purposes.

The only other unusual use of the stack is that the reserved identifiers are declared as the first stack elements in the global stack frame. These values must be initialised by the interpreter before program execution.

6.3 The heap

There are three heap objects strings, vectors and structures. The empty string is represented by a nil pointer on the pointer stack so that the heap does not require special arrangements for them. The instructions which manipulate the strings must take account of this. There is no way in which a uninitialised vector or structure can be obtained in the language. The manner in which the heap items are represented is as follow

Strings

word 0

bits 14-15 10	if a string
bit 13 0	
bit 12	mark bit for the garbage collector
bits 0-11	number of characters in the string

This is followed by the string elements one to a byte with the last byte spare if there are an odd number of characters.

Vectors

word 0

bits 14-15	00	if vector
bit 13	1	if elements are pointers
bit 12		mark bit for garbage collection
bits 0-11		size of the vector on the heap

word 1 lower bound

The upper bound is found from the equation

$$size = (upb - lwb + 1) * 2 + 4$$

Structures

word 0

bits 14-15	01	if a structure
bit 13	1	if it contains pointers
bit 12		mark bit for garbage collection
bit 0-11		trademark

The structure implementation is perhaps the most clever of the three. The aim is to reduce the overhead of housekeeping information on each structure. However, the structure must carry its trademark for structure class checking, an indication of which fields are pointers for structure creation and garbage collection and its size on the heap also for garbage collection. The trademark is kept with the structure and is used as a unique pointer for that structure class into the structure table. The structure table contains the size of the structure on the heap and the number of pointers it contains for each structure class. Unknown to the user the compiler arranges the pointer fields to be at one end of the structure. Thus, the garbage collector can find them all by merely knowing

how many there are. It can also find the size of the item indirectly from the trademark. The number of pointer fields is also required by the form structure instruction to fill the fields from the correct stack. This arrangement keeps the amount of information stored with each structure incarnation to a minimum.

6.4 Free Space List

Initially the heap is empty. Requests to form vectors, strings and structures use up the heap space. Garbage collection frees this space when it is no longer in use. If the garbage collector is non compacting, as in the PDP11 case, the store may become fragmented. That is, the free store may not be in one contiguous block but interspersed with allocated heap storage. When this is the case a quick method of identifying the free store is required since it is this free store that will be used when new objects are formed.

The PDP11 implementation of the S-algol heap links the blocks of free store together to form the free space list (FSL). Every block of free store contains two words at the beginning which contain

a. A pointer to the next block of free store or an indication that this is the last block on the chain

b. The size of the block

Every free block on the heap must be at least two words in length. Single free words are marked as such and not kept on th FSL. However, since they can be recognised , they may be coalesced into a larger block during garbage collection.

When a piece of store is requested, the free space list is

searched using a first fit algorithm. If no block of the required size is found a garbage collection is performed. If the space still cannot be found the program is terminated as it is out of store. Otherwise a block of store has been found. There are three possibilities

a. The block fits exactly

b. The block is one word too big

c. The block is two or more words too big

In the first case the links in the FSL must be altered to remove this block. In the second case the links must be altered in the same way and the spare word marked appropriately. In the third case the block looks like this

The requested section of the free block is taken from the top end of the block. This means that only the size field in the free block need be updated. The free list is held in address order.

6.5 Marking the Heap

The first phase in collecting the unused space on the heap is to identify the used space. This is done by taking each element of the active pointer stack and following all the pointers recursively. The PDP11 version takes the pointer from the pointer stack and places it on the main stack. The element that it points to is then marked and all the pointers in the element placed on the main stack overwriting the original pointer. This process is repeated until no pointers are left on the main stack. Of

course, any item that is encountered that has already been marked need not be followed again. When finally all the pointer stack items have been used, the marking phase is complete.

The system works well for large linked lists since only one pointer at a time will be on the main stack. Vectors with pointer elements may cause size problems on the main stack if they are too big. However, it should be possible to write a marker that handles these two items differently.

String literals on the PDP11 version are kept with the program code. These will be marked. This could be avoided by checking that the element being marked is in the heap. But, it is of no consequence since nothing apart from the garbage collector looks at the mark bit.

6.6 Garbage Collection

Once the heap has been marked it is the job of the collection phase to release the unused space on to the free list. This is achieved by looking at every heap item in turn. The item can be in one of four states

a. On the free list

b. A single free word

c. Allocated but not marked

d. Allocated and marked

The heap and the free list are scanned simultaneously in address order. If the block under consideration is marked, the garbage collector unmarks it and skips over it. It was explained earlier how the size of the block is found. If the block is on the FSL it is skipped over. Blocks freed by the garbage collector are placed on the free list. As this is being done the blocks are coalesced into larger blocks if possible by looking at the

adjacent blocks. Any free single words may also be made part of a larger block if possible.

This method of garbage collection is extremely simple and very efficient in performance. Of course, since the collector does not compact the free space, it takes longer to allocate store and some requests may not be granted even although there was enough free store available. It is felt that the complexity of writing a compacting garbage collector does not balance the advantages. The PDP11 free store allocator and garbage collector is about 128 PDP11 instructions which is very compact.

6.7 The Input Output System

Files are first class data objects in S-algol. However, there is no method of creating an object of type file in the language. It is left to the individual environment to provide the facilities for file manipulation. In UNIX, the file system allows the user to create file dynamically. This is reflected in the facilities to use files in the UNIX version of S-algol.

There are five standard functions which use files.

```
procedure open( string name ; int mode -> file )

procedure close( file F )

procedure seek( file F ; int offset,ptrname )

procedure eof( file F -> bool )

procedure create( string name ; int mode -> file )
```

The use of these functions should be fairly obvious.

To cut down the number of system calls the I/O is buffered. The I/O descriptor is a pointer to a vector on the heap which contains the

buffer. This buffer contains

a. The number of items left

b. A pointer to the next item

c. A pointer to the start of the buffer

d. An end of file indicator

e. A read/write indicator

f. A 512 byte buffer

The file may be a read file, a write file or both. If it is both then synchronisation can become a problem and care has to be taken to overcome this. Before any read is performed on these files a write is executed if necessary. This gives the illusion of unbuffered I/O to the user with the advantages of buffering to the system.

7. System Measurement

The measurement of any computer system should take place throughout the lifetime of the system. Since the S-algol system is still being developed the measurement is not complete. However, a number of software tools have been built to enable an initial assessment of the implementation technique. These tools are considered only as a basic set and are not meant to be exhaustive. They have been chosen carefully to give a reasonable picture of the real system.

7.1 Flow Summary

The first tool is that of a flow trace summary. The user can indicate to the compiler that this is required and when the program is run information is collected on how many times each line of source code is executed.

The user can then have this information printed out and an example of the facility is given in Appendix II. From the summary it can be seen where the program spends its time. Knuth [56] has suggested that 85-90% of the time taken in a program comes from 5-10% of the code. The flow summary facility allows the programmer to find that 5% and optimise if necessary. This, of course, is extremely useful in the assessment of any algorithm since it is not always obvious what the results will be.

When applied to the compiler itself the areas of source code which were executed the most were parts of the lexical analysis and the environment handler. Using this information an improvement which is discussed later was made to the compiler.

7.2 Static Code Analysis

This tool was built to allow the implementor to assess the effectiveness of the instruction codings with regard to space. It was originally built for the PDP11 interpretive system but could be easily applied to a code generation system as well. The program will analyse the code output by the compiler and print out the number of occurrences of each instruction. The space taken up by the operation codes and the data is then calculated followed by a Huffman [55] coding to give a theoretical minimum.

This tool allows the implementor to experiment with different instruction encodings and calculate their effect on the code size. Predictions can also be made on how much space an implementation, interpretive or code generative, will use.

7.3 Dynamic Instruction Analysis

The final tool that was developed to measure the S-algol system was a dynamic instruction count. Every time an abstract machine instruction is executed, the count for that instruction is incremented. By weighting each instruction according to its execution time, an estimate of the total system run time can be made. Altering one of the instruction weightings will give a new estimate for the total time. Therefore, this may be used to identify the most commonly used instructions and to measure the effect on the speed of the system in improving their implementation. This tool is again only of interest to the implementor.

7.4 Results

Appendix II gives the results of all three measurement tools on a smallish program. To the user, only the flow summary is of real use. However, to the implementor of the S-algol system all three tools would be

used together. Wilner [54] suggests that only first order improvements are worth achieving in altering implementations. Thereafter a law of diminishing returns applies. For greater effect the basic source algorithms of the program must be improved. The advantage of the three S-algol tools is that they identify the source areas of interest as well as giving information on the effectiveness of the improvements.

The measurement system was applied to the compiler itself and a number of results obtained. Since the compiler is still being improved the figures quoted are valid only at the time of measurement and do not necessarily reflect the position at the present time. However, when comparisons have been made care has been taken that like is being compared with like.

The compiler, on compiling itself, was first subjected to a flow summary. One of the areas which was found to be heavily used was the environment handler. At this time the environment was held as a simple linked list with a list superimposed on it to mark the blocks. The static size of the program was 9492 words with 12,897,438 interpretive instructions being executed when the program was run. The environment representation was altered to a linked list of binary trees as described in Chapter 4. The result was that the compiler size remained the same, a coincidence, but the dynamic count was reduced to 8,541,602 interpretive instruction executions. This is an improvement of 33.8%.

The S-algol case clause allows the user to specify the order in which to look for a match. By ordering the case selectors in order of dynamic frequency, the total number of instructions for the above system was further reduced to 7,745,125 instructions. This is a further reduction of 9.4%. This is a typical result of using the measurement system to improve the source code. However, it should be remembered that the changes

refer to the dynamic evaluation of one program with fixed data and may not be appropriate in all cases.

The static analysis is used to improve the store compaction. In one version of the compiler it was found that there were 903 procedure calls. The code for a procedure call consisted of

```
load.closure
mark.stack
evaluate parameters
apply
```

This version of the compiler consisted of 10813 words. By combining the load.closure and mark.stack instructions a saving of 903 words is made. This is an 8.4% reduction in space. This meant a speed increase as well since only one instruction has now to be decoded.

A Huffman coding was also performed on the S-code for the compiler. The result was that a minimum of 43,096 bits were required to represent the operation codes in the compiler. The initial guess at instruction formats of 4,6 and 9 bits required 58,545 bits which is 73.6% of the optimum. Wilner suggests that a result as good as this constitutes a good initial estimate.

Finally, in Chapter 5 an assumption on the distribution of names between local, global and intermediate environments was used in the basic design of the S-code machine. The static and dynamic counts for the instructions which access these values is now given for one version of the compiler.

Static count

local	315	global	234	load	9
localaddr	35	globaladdr	101	loadaddr	6
plocal	599	pglobal	636	pload	12
plocaladdr	61	pglobaladdr	35	ploadaddr	4
dlocal	10	dglobal	3	dload	0
mst.local	76	mst.global	813	mst.load	14
Total	1096		1822		45
%	37.0%		61.5%		1.5%

Dynamic count

local	535,031	global	594,308	load	1,037
localaddr	58,679	globaladdr	145,219	loadaddr	303
plocal	888,766	pglobal	696,610	pload	3,406
plocaladdr	36,276	pglobaladdr	71,540	ploadaddr	1,001
dlocal	0	dglobal	0	dload	0
mst.local	8,675	mst.global	325,400	mst.load	180
Total	1,527,427		1,833,077		5,927
%	45.4%		54.5%		0.1%

The assumption that most names are either local or global is borne out by the above results, but this may merely reflect the style in which the compiler was written.

In the final analysis these tools have proved very useful and effective. Part of their strength derives from the fact that they not only measure an existing system but can also be used to predict the effect of an alteration. However, further work must be performed before any major claims can be made.

8. Conclusions

At the end of any piece of work an attempt should be made to assess its worth. Realistically , quantum jumps in research are rarely achieved and more often small additions to knowledge are made. However, it is from the base of these small additions that the quantum jumps are made. So much for expectation. What has been achieved?

It was observed that our general purpose programming languages were becoming intellectually unmanageable. Progress in language design had been equated with continuously producing innovations without much emphasis on how they could be used. It was therefore felt that an attempt to place language design on a firmer basis was necessary. Of course, it is of little use proposing a methodology of language design without being prepared to put it to use. A language in the algol tradition, called S-algol, was designed under the proposed rules and implemented. This thesis describes that design methodology and the design and implementation of S-algol.

The success of the modern programming languages is usually dependent on the balance between the descriptive power it allows and the efficiency that is achieved on the target machine. The flavour of the language depends on the emphasis the implementor places on either aspect. Unfortunately this has led to confused and large languages, since a decision by an implementor to restrict some facility on efficiency grounds is not always clear to the user. The design technique confuses the two issues and succeeds in satisfying neither.

This work has attempted to clarify the design process. One concession to efficiency is made at the start. That is, the languages of interest are the algols and in particular they have conceptually infinite reusable stores.

From then on the design process concentrates on the descriptive

power of the language, returning later to the implementation details. This is not the end of the story however. Merely removing the implementation in part from the design of the language does not guarantee a good language. Therefore a design methodology is proposed for use.

The methodology is based around three rules which are well known to workers in the field of semantics. They are

a. The principle of correspondence

b. The principle of abstraction

c. The principle of data type completeness

Hitherto these rules have been used in the analysis of languages to highlight their differences and shortcomings. Here they are used in the synthesis of a programming language with the aim that they simplify and somehow make the language more intellectually manageable. This simplicity is not achieved at the expense of power, indeed more power is derived from the simplicity or rather from the lack of restriction.

Just as the implementation and design have been partially separated for clarity, it is felt that the syntactic and semantic issues of the language should be initially separate. Again this is not done to downgrade one of the issues but rather to avoid unnecessary confusion. The semantic domains of the language are first proposed and subjected to the above rules and then a light coat of syntax is applied.

The language that is the result of all this careful design is S-algol. In what way is the language new or better? What are the innovations?

First of all, S-algol is not without innovations. In particular, the strings as a scalar data type, only allowing initialising declarations and call by value parameters, the vectors as first class citizens, the case

clause, the mixing of clauses and declarations and introducing constancy as a protection issue are all new. Although these ideas help, the strength of the language does not derive from them. It comes from the fact that it is carefully designed from a set of rules that allow no exceptions to the general case. Thus, at least S-algol can be assessed scientifically using these ground rules. Around this castle, a light syntax has been thrown to enhance its beauty.

The language has only four semantically meaningful syntactic categories, clauses typed and untyped, declarations and sequencers which is about the same as most other algols. Therefore there is no reduction of power here. The declarations which only allow values to be named are designed strictly in conjunction with the call by value parameters. This greatly simplifies the language and makes it safer without loss of power.

Have the rules been broken? The answer is that they have. In the case of abstractions over declarations and sequencers it is felt that other criteria can be used to justify their exclusion. However, it should be noted that the technique used to exclude something is new. There is still a general rule, now with two exceptions, not a whole host of particular rules. There are no exceptions to the principle of correspondence or the principle of data type completeness.

The implementation technique for S-algol is also investigated. A compilation system loosely based on the structuring method of Ammann is developed. The layers of abstraction that comprise the compiler are quite different however, and it is noted that the size of the compiler owes much to the fact that these layers are well chosen. The compiler is written in S-algol itself with the machine dependent parts being isolated as far as is possible. This isolation is never completely successful.

The strength of the recursive descent technique is not that it

gives faster compilers, for who cares in a world of excess processor power, but that it makes the compilers smaller and easier to check and maintain. It also gives more portable compilers since the portability depends on the number of alterations to the source code, which is some function of the total size. Therefore work in the development of this technique has been worthwhile from a software engineering point of view.

The project lays no claims to the enhancement of the theoretical properties of LL parsing or even recursive descent but has concentrated on the software engineering aspects. To this end, an improvement on the methodology of the previous work is proposed and applied. Hopefully others will be able to develop it further.

An abstract machine, the S-code machine, is also designed and implemented. The first machine of this style was the beta machine of Randell and Russell. The S-code machine is a distant relation.

A method of procedure entry and exit which does not use a display is invented. It is based on the principle that most stack items are either local or global in scope. This is certainly true of the compiler as the measurement statistics of Chapter 7 show. The S-code machine has a heap with a garbage collector since not all data items in the language can be implemented on a stack. The problem of identifying the transient pointers on the stack when a garbage collection struck led to the invention of the second stack for pointers.

How then can the project be assessed? Certainly the implementation techniques developed are powerful and easy to use. Enough confidence in them has allowed expansion to other machines such as the IBM 370, Intel 8086 and the Zilog Z80. It has also dispelled a number of fears in writing compiler systems and in systems with garbage collectors. Therefore, it is reasonable to claim an advance in software technology.

It is much more difficult to assess a new language or design methodology. The principle design tenet is simplicity - power through simplicity, simplicity through generality. However, with complete generality comes chaos. For example, given a piano and complete freedom in tuning, it is almost impossible to say anything sensible about the music. A standard tuning is proposed. However, this may go to the other extreme and only allow trivial notions to be expressed. It is with this balance between the power of generality and its lack of security that language designers are concerned.

A design methodology based on semantic principles has been proposed. It was then used in the synthesis of S-algol. The methodology is concerned with achieving a balance between power and security but can only be considered as a first step. Hopefully it will provoke further discussion and be developed further itself. Since programming and programming languages are still relatively badly understood, it is difficult to hope for more. It is at least an attempt to systematise the process of language design and provides a technique for comparing languages.

The language S-algol is itself simple and easy to use. It has the power of an algol between Algol 68 and Algol W. However, it has far fewer rules and exceptions than either of these and is therefore simpler to understand and teach. At first it was proposed as the base language in a family. Now it is presented as a member of a family which is defined by the rules of the design methodology.

References

1. Landin P.J., The next 700 programming languages
CACM Vol 9 No 3 pp 157-164 March 1966

2. Wirth N., On the design of programming languages
IFIP Congress 1974 North-Holland 1974

3. Wirth N., PL360 A programming language for the 360 computer
JACM Vol 15 P 36 1966

4. McCarthy J.,et al Lisp 1.5 Programmers manual
M.I.T. Press Cambridge Mass. 1962

5. Turner D.A., SASL language manual
University of St.Andrews CS/79/3

6. Naur P.,et al Revised report on the algorithmic language Algol 60

7. Wirth N.,Hoare C.A.R., A contribution to the development of algol
CACM Vol 9 No 6 pp 413-431 June 1966

8. Wirth N.,Weber H., EULER, a generalisation of algol
CACM Vol 9 Nos 1,2,12

9. Morrison R., The Algol R reference manual
University of St.Andrews CS/78/1

10. Wirth N., The programming language Pascal
Acta Informatica Vol 1 1971 pp 35-63

11. van Wijngaarden A.,et al Report on the algorithmic language Algol 68
Numerische Mathematik Vol 14 1969

12. IBM Corporation System /360 PL/1 Language reference manual
Form GC28-8201-4 1972

13. Dijkstra E.W., The humble programmer
CACM Vol 15 No 18 Oct 1972

14. Habbermann A.N., Critical comments on the programming language Pascal
Acta Informatica Vol 3 PP 47-57 1973

15. Dijkstra E.W., Goto statement considered harmful
CACM Vol 11 147-8,538,541 1968

16. Lecarme O.,Desjardins P., More comments on the programming language Pascal
Acta Informatica Vol 4 pp231-243 1975

17. Minimal Basic
European Computer Manufacturers Association ECMA-55

18. Iverson K.E., A programming language
Wiley 1962

19. Addyman A.M.et al A draft description of Pascal
Software,Practice and Experience Vol 9 p 381-424 1979

20. van Wijngaarden A., Generalised algol
Annual Review of automatic programming No 3 1963

21. Curry H.B.,Feys R., Combinatory logic
North-Holland 1958

22. Johnston J.B., A contour model of block structured processes
ACM Sigplan Notices Vol 6 No 2 1971

23. Henderson P., An approach to compile time type checking
Information Processing 1977 North-Holland

24. Tennent R.D., Language design methods based on semantic principles
Acta Informatica Vol 8 pp97-112 1977

25. Strachey C., Towards a formal semantics
Formal language description languages North-Holland 1966

26. Quine W.O., Set theory and its logic
Harvard University Press 1963

27. Donahue J.E., Locations considered unnecessary
Acta Informatica Vol 8 pp221-242 1977

28. Hoare C.A.R., Recursive data structures
Int. J. of Computer and System Sciences Vol 4 pp105-132

29. Hehner E.C.R., On removing the machine from the language
Acta Informatica Vol 10 pp229-243 1978

30. Ichbiah et al Rationale of the design of the programming language Ada
Sigplan Notices Vol 14 No 6 June 1979

31. Strachey C., Fundamental concepts in programming languages
Oxford University 1967

32. Richards M., BCPL, a tool for compiler writing and systems programming
AFIPS SJCC 1969

33. Gunn H.I.E.,Morrison R., On the implementation of constants
Information Processing Letters Vol 9 No 1 July 1979

34. Hoare C.A.R., A note on the for statement
BIT Vol 12 pp334-341 1972

35. Schuman S.A., Towards modular programming in high level languages
Algol Bulletin 37 pp12-23 1974

36. Legard H.F.,Marcotty M., A genealogy of control structures
CACM Vol 18 No 11 1975

37. Knuth D.E., Structured programming with goto statements
Computing Surveys Vol 6 pp261-301 1974

38. Morrison R., An algol compiler for the PDP11 computer
Proc. U.K. DECUS Conference 1979 Canterbury

39. Lindsay C.H., Modals
Algol Bulletin 37.4.3

40. Griffiths M., LL(1) grammers and analysers
Advanced Course in Compiler Construction Munich 1974

41. Richards M., The portability of the BCPL compiler
Software, Practice and Experience Vol 1 p135 1971

42. Bauer H.,Becker S.,Graham S., AlgolW implementation
Technical Report No CS98 Stanford University

43. Randell B.,Russell L.J., Algol 60 implementation
Academic Press 1964

44. Ammann U., The development of a compiler
Proc. Int. Symposium on Computing North-Holland 1973

45. Turner D.A.,Morrison R., Towards portable compilers
University of St.Andrews TR/76/5

46. Backhouse R., Syntax of programming languages
Prentice-Hall 1979

47. Turner D.A., Error diagnosis and recovery in one pass compilers
Information Processing Letters Vol 6 No 4 1977

48. Hauck E.A.,Dent B.A., Burroughs B6500/6700 stack mechanism
AFIPS SJCC 1968

49. Nori K.V.,et al The Pascal P compiler implementation notes
Technical Report No 10 Zurich

50. Wichmann B., Algol 60 compilation and assessment
Academic Press 1973

51. Dahl O.J.,Dijkstra E.W.,Hoare C.A.R., Structured programming
Academic Press 1972

52. Morrison R., A method of implementing procedure entry and exit
Software, Practice and Experience Vol 7 1977

53. Ritchie D.M.,Thompson K., The UNIX timesharing system
CACM Vol 17 No 7 July 1974

54. Wilner W.T., The design of the B1700
Burroughs Corporation

55. Huffman D., A method for the construction of minimum redundancy codes
Proc. IRE Vol 40 1952

Appendix I

S-algol

Reference Manual

56. Knuth D.E., An empirical study of Fortran programs
Software, Practice and Experience Vol 1 No 2 1971

57. Dijkstra E.W., Guarded commands, nondeterminacy and formal derivation
CACM Vol 18 No 8 1975

58. Foster J.M., A syntax improving device
Computer Journal May 1968

Contents

Chapter

Appendices

1. Introduction

Programming language design is probably the most emotive subject in Computational Science today. Nearly everyone uses a programming language and most people have something to say about their design.

S-algol is presented as a member of a family of languages in the algol tradition that are constrained by a design methodology. This design methodology is based on the belief that most of out programming languages are too big and intellectually unmanageable. In addition it is believed that these problems arise in part from the languages being too restrictive. The number of rules to define a language increases when a general rule has additional rules attached to to constrain its use in certain cases. Ironically these additional rules usually make the language less powerful. Power through simplicity, simplicity through generality is the guiding tenet.

The design methodology is based on three semantic principles which can be attributed to Strachey [1]. These are

a. The principle of data type completeness

b. The principle of abstraction

c. The principle of correspondence

The principle of data type completeness states that all data types must have the same "civil rights" in the language and that the rules for using data types must be complete with no gaps. This does not mean that all operators in the language need be defined on all data type but rather that general rules have no exceptions. Examples of lack of completeness can be seen in algol W [2] where arrays are not allowed as fields of records and in Pascal [3] where only some data types are allowed as members of sets. This principle is bound to lead to simpler languages since it

avoids the complexity of special cases.

Abstraction is a process of extracting the general structure to allow the inessential details to be ignored. It is a facility well known to mathematicians and programmers since it is usually the only tool they have to handle complexity. The principle of abstraction when applied to language design is invoked by identifying the semantically meaningful syntactic categories in the language and allowing abstractions over them. The most familiar form of abstraction is the function which is an abstraction over expressions.

Finally, the principle of correspondence states that the rules for introducing and using names should be the same everywhere in a program. In particular there should be a one to one correspondence between introducing names in declarations and introducing names as parameters.

Armed with the above rules the language may be designed as follows

a. Data Types

Decide which data types, both simple and compound, are required by the language and define the operations on these data types. The flavour of the language will be determined by the data objects it can manipulate. The principle of data type completeness is invoked to ensure that all data objects have the same civil rights. Ignoring the principle means introducing rules to handle exceptions thus making the language more complex. For example, if arrays are allowed then arrays of arrays should be allowed as should expressions and functions with array results.

b. The Store

Introduce the store, if any, and the manner in which it may be used. First of all the relationship between the store and the data types

should be defined. This includes the implementation of pointers, data locations and protection on these locations. For example, constants may be regarded as storage locations which may not be updated [7].

The introduction of the store forces consideration of the language control structures. The designer can take his pick and is well advised to read the excellent paper by Legard and Marcotty [5] on the subject.

c. Abstraction

Tennent [6] has suggested that the method of applying the principle of abstraction is to identify the semantically meaningful syntactic categories and invent abstractions for each. This he does for Pascal and proposes some extensions to complete the abstractions. However, he points out that it is not always an easy matter to identify these categories in the first place. Most languages have at least the following.

Syntactic category	Abstraction
expression	function
statement	procedure
declaration	module
sequencer	sequel

Functions and procedures should be familiar. The name module derives from Schuman [4] and the sequel from Tennent.

The problem is to identify the useful abstractions. For example, for those of us in love with the algol scope rules, the module is a peculiar abstraction especially since the same power can be derived from function producing functions. Similarly the sequel looks like another convoluted goto.

d. Declaration and Parameters

Invent the declarations and the parametric objects together. There must be a one to one correspondence between the two. This does not mean that they necessarily have the same syntax but that for every type of declaration there is an equivalent parametric type. Parameter passing modes are also included in this correspondence. For example, the declarative equivalent of call by value is an initialising declaration. If functions, record classes etc can be declared then they can be passed as parameters even though they are not defined as data objects in the language. Finally, if the language has a facility to define new data types and give them a name then the type can be passed as a parameter rather like algol 68 modals [8].

e. Input and Output

The I/O models for most high level languages tend to reflect the environment in which they were designed. Some attempts have been made to design and implement comprehensive I/O systems. Unfortunately where it has not been tied to particular hardware it has never been very successful. Nowhere else in the design of a programming language does the hardware intervene as much as it does in the I/O system. When a new I/O device becomes available the language must be able to make use of it. Of course, this situation is hopeless and perhaps the wisest approach to I/O is to allow the implementor to deal with it for a particular environment, as the Algol 60 designers proposed.

f. Iterate

Re-evaluate the language and correct or justify any idiosyncrasies in the design. Hopefully the design process will converge.

g. Concrete Syntax

The final stage of language design is to propose a concrete syntax. Ideally different groups of workers could have a different syntax. However, there are many users who do not wish to design their own syntax and so the language must provide at least one possibility.

It seems very obvious to say that the syntax should be simple and easy to learn. That may be so but there is no doubt that some of the success of the language depends on the cosmetics. Also, a carefully chosen syntax can ease the problem of compilation.

How often the rules can be broken is for the designer's own conscience. However, every time the rules are broken the language becomes more complex. The rules were introduced to help design simpler and more intellectually manageable languages and should only be ignored with great care.

S-algol is the first of the family of algols to be produced under this design methodology. It reflects my own personal preference for data objects and syntax. It also ignores the principle of abstraction with regard to sequencers and declarations. Sequencers because they are obnoxious and declarations because it breaks with the algol tradition of scope rules. However there are no exceptions to the principle of correspondence or the principle of data type completeness in the language. I see this as the strength of the language.

A number of people must be thanked for their help with the S-algol project. Firstly, Professor Jack Cole whose enthusiasm and encouragement were responsible for the development of the system. Peter Bailey must be thanked for his help in testing the system and in proof reading, as well as his enormous willingness to help when difficulties arose. Paul Maritz, who is currently implementing S-algol on the Zilog Z80, pointed out a

discrepancy in the application of the principle of correspondence to the language. Mike Livesey must also be thanked for our discussions of the semantic principles on which the language is based and David Turner, now in Canterbury, with whom I started on language design deserves mention for his contribution. Whether they like it or not their influence is present in S-algol.

These people have contributed to the language design as have the useful comments of Hamish Gunn, Tony Davie, Michael Weatherill and Ian Sommerville at Strathclyde University. My special thanks to Tony for his SPELL command and to the person who wrote the UNIX roff program with which this manual was prepared.

2. Syntax Specification

The syntax of S-algol is given in two parts

a. a context free grammar written in BNF

b. a context sensitive set of rules to govern the possible types of expression in the language.

These sets of rules are short and transparent and define the legal class of programs as briefly as possible. Wherever it is felt necessary, English narrative has been added to clarify some detail. This method of language description, first used by Turner [9] is simple and easy to use and indeed was the method used to define the syntax in the first instance.

2.1 BNF Extensions

In order to improve the readability of the S-algol syntax, two commonly used extensions to BNF have been used. The square brackets "[" and "]" are metasymbols which are used when a syntactic object is optional.
e.g.

< scale factor > ::= [< addop >]< unsigned integer >

can be read as

"a scale factor consists of an optional addop followed by an unsigned integer".

The metasymbol "*" is used to signify that an object may be repeated zero or many times.

e.g.

< unsigned integer > ::= < digit >[< digit >]*

can be read as

"an unsigned integer consists of a digit which may be followed by any number of digits".

Using BNF and the two extensions above the complete context free syntax of S-algol can be described in about 40 productions. It is given in Appendix A.

2.2 Type Matching Rules

For an S-algol program to be syntactically correct it must obey both the BNF and the type matching rules. The legal data types in S-algol are given later, but the notation for describing the type rules is introduced here. By keeping the two sets of rules distinct, the program forming rules are easier to follow.

For each BNF production in the language there is an equivalent type matching rule which further qualifies the manner in which the syntactic construct may be used. In a production where any of the non-terminals has a data type associated with its legal use, the type rule has that data type name enclosed in the meta symbols "{" and "}" instead of the non-terminal. The type rule also indicates the result type of the production by use of the meta symbol "=>".

e.g. the or expression

BNF < expression > or < expression >

TR { bool } or { bool } => { bool }

The BNF indicates that an expression may be formed by taking two subexpressions and applying the operator or to them. The type rule (TR) further qualifies this by showing that both subexpressions must be of type bool and that the result is of type bool. Anything else is illegal.

Further

BNF if < clause > then < clause > else < clause >

TR if { bool } then { T } else { T } => { T }

These rules indicate the form of the if clause and show that the first clause must be of type bool. The two alternatives can be of any type but must be the same, producing a result of that type.

Thus if a < b then true else 4 is not permitted

whereas if a < b then 4 else 6 is valid and of type int

A full presentation of the type rules is given in Appendix B. Throughout this manual the type rules will be used to clarify legal statements in the language rather than a formal definition.

3. Types and Data Structures

There are an infinite number of data types in S-algol defined recursively by the following rules.

a. The scalar data types are int, real, bool, string and file

b. For any data type T, *T is the data type of a vector with elements of type T.

c. The data type pntr comprises of a structure with any number of fields, and any data type in each field

In addition to the above data types there are a number of other objects in S-algol where it is convenient to give them a type in order that the compiler may check their use for consistency.

d. The type of a procedure with parameters T1,.....,Tn and result type Tm is (T1,.....,Tn -> Tm)

e. Clauses which yield no value are of type void

f. The class of a user defined structure with fields of type T1,....Tn is of type (T1,.....,Tn)-structure and its fields are of type Ti-field

The user requires to know about these types in order to follow the complete type matching rules.

4. Literals

Literals are one of the basic building blocks of a program and allow values to be introduced.

4.1 Integer Literals

These values of type int are defined by

< integer literal > ::= < digit >[< digit >]*

An integer literal is at least one digit followed by any number of digits.

e.g. 1 0 1256 8797

Notice that negative literals are not allowed. However they can be written down as expressions.

4.2 Real Literals

These are of type real and are defined by

< real literal > ::= < integer literal >[.< integer literal >][e< scale factor >]

< scale factor > ::= [< addop >]< integer literal >

< addop > ::= +|-

Thus, there are a number of ways of writing a real literal

e.g. 1.2 3.1e2 5e5

1. 3.4e-2 3.4e+4

3.1e-2 means 3.1 time 10 to the power -2 i.e. 0.031

4.3 Boolean Literals

There are only two literals of type bool. They are the symbols true and false. They may be used, with obvious meaning when a boolean literal is required.

4.4 String Literals

< string literal > ::= "[< char >]*"

A string literal is any sequence of characters in the set (hopefully ascii) enclosed by double quotes. The empty string is denoted by "". Examples of string literals are

"This is a string literal"

"I am a string"

The programmer may wish to have a double quote itself inside a string literal. This requires using a single quote as an escape character and so if a single or double quote is required inside a string literal it must be preceded by a single quote.

e.g. "a'"" has a value a" and

"a''" has a value a'

There are a number of other special characters which may be used inside string literals. They are

'n	newline
'p	newpage
'o	carriage return
't	horizontal tab
'b	backspace

These characters are normally used in input and output.

5. Expressions and Operators

The general form of an S-algol expression is

< expression > ::= < unary operator >< expression >|
< expression >< binary operator >< expression >|
(< expression >)

5.1 Arithmetic Expressions

Arithmetic may be performed on data objects of type int and real. The operators are

< unary operator > ::= +|-

< binary operator > ::= +|-|*|/|div|rem

They mean

+	addition
-	subtraction
*	multiplication
/	real division
div	integer division throwing away the remainder
rem	remainder after integer division

Some examples of arithmetic expressions are

a + b 3 + 2 1.2 + 0.5 -2 + a/2.0

5.2 Precedence Rules

The order of evaluation of an expression in S-algol is from left to right and based on the precedence table

* / div rem
+ -

That is, the operations * / div rem are always evaluated before + and -. However, if the operators are of the same precedence then the expression is evaluated left to right.

e.g.

6 div 4 rem 2 gives the value 1

Brackets may be used to override the precedence of the operator or to clarify an expression.

3 * (2 - 1) yields 3 not 5

5.3 Expression Types

The rules for determining the type of the result of an expression are

< unary operator > { T } => { T]

where T is int or real.

That is, a unary operator followed by an expression of type real or int produces a result of that type.

e.g. -3.2 is of type real

+432 is of type int

Bracketing does not alter the expression type.

i.e. ({ T }) => { T }

The rules for the arithmetic binary operators are

{ int } + { int } => { int }

{ int } - { int } => { int }

{ int } * { int } => { int }

{ int } div { int } => { int }

{ int } rem { int } => { int }

{ real } + { real } => { real }

{ real } - { real } => { real }

{ real } * { real } => { real }

{ real } / { real } => { real }

In S-algol whenever an integer is used where a real is required the compiler will coerce the integer into a real and give the result as real. Thus

{ int } + { real } => { real }

Examples

3.0 + 1	is of type real
3 div 2	is of type int
3.0 rem 4	is illegal

The successive application of the type rules in accordance with the precedence rules will produce the type of any S-algol expression.

5.4 String Expressions

There is only one string operator ++ defined on strings. It concatenates the two operand string to form a new string.

e.g.

"abc" ++ "def" results in the string "abcdef"

The type rule is

{ string } ++ { string } => { string }

A new string may be formed by selecting a substring of an existing string.

e.g.

if s is the string "abcdef"

then s(3|2) is the string "cd"

That is a new string is formed by selecting two elements from s starting at element 3.

BNF < name >(< clause >< bar >< clause >)

TR { string }({ int }|{ int }) => { string }

For the purposes of substring selection the first character in a string is numbered 1. The selection values are the start position and the length respectively.

5.5 Boolean Expressions

Objects of type bool in S-algol can have the value true or false. There are only two boolean literals, true and false and three operators. There is one unary operator, ~, and two binary operators, and and or. They are defined by the truth table below.

a	b	~a	a or b	a and b
true	false	false	true	false
false	true	true	true	false
true	true	false	true	true
false	false	true	false	false

As with the arithmetic operators the precedence is important and is defined as

~

and

or

Thus ~a or b and c is equivalent to (~a) or (b and c). The type matching rules for the operators are

~{ bool } => { bool }

{ bool } and { bool } => { bool }

{ bool } or { bool } => { bool }

Finally, the evaluation of a boolean expression in S-algol is non-strict. That is in the left to right evaluation of the expression, as soon as the result is found, no more computation is performed on the expression.

e.g.

true or < exp >

gives the value true without evaluating < exp > and

false and < exp >

gives the value false without evaluating < exp >.

5.6 Relational Operators

Expressions of type bool can also be formed by some other binary operators. For example, a = b is either true or false and is therefore boolean in nature. The operators are called the relational operators and are

<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
=	equal to
~=	not equal to

The type rules are

{ T }< rel.op >{ T } => { bool }

where < rel.op > ::= <|>|<=|>=|=|~=

and T is of type int, real or string consistently.

Furthermore, = and ~= are defined on all data type in S-algol.

{ T } = { T } => { bool }

{ T } ~= { T } => { bool }

5.7 Precedence Table

The full precedence table for S-algol is now

/ * div rem

+ - ++

~

= ~= < <= > >= is isnt (see structures)

and

or

6. Names and Declarations

In S-algol a name may be given to a data object, a procedure, a procedure parameter, a structure field and a structure class. A name is called an identifier and may be formed by the syntactic rule

< identifier > ::= < letter >|< identifier >< letter >|
< identifier >< digit >|< identifier >.

That is, an identifier consists of a letter followed by any number of dots, letters or digits.

e.g.

x1 ron look.for.record1

6.1 Declaration of Data Objects

Before a name can be used in S-algol it must be declared. The action of declaring a name associates a location of a certain type which can hold values that the name may take. In S-algol the programmer may specify whether the value is constant or variable. A constant may be manipulated in exactly the same manner as a variable except that it may not be updated.

When introducing a name the programmer must indicate the identifier, the type of the data object, whether it is variable or constant and its initial value. By forcing the user to specify an initial value one type of error, that of an uninitialised name, is completely eliminated and need not be checked for.

Variables are declared by

let I := E

where I is an identifier and E is a clause resulting in a data value.

e.g.

```
let a := 1
```

introduces an integer variable with initial value 1. Notice that the compiler can deduce the type.

A constant is declared by

```
let I = E
```

e.g.

```
let discrim = b * b - 4.0 * a * c
```

introduces a real constant with the calculated value. The compiler will detect and flag as an error any attempt to alter a constant.

7. Compound Data Objects

S-algol allows the programmer to group together data objects into larger compound objects which may then be treated as single objects. If the constituent objects are of the same type a vector is used and a structure otherwise. The vectors and structures have the full "civil rights" of any other data object in S-algol. However, from an efficiency point of view it is unwise to copy these objects on assignment. To overcome this, objects of type *T or pntr are regarded as pointers to vectors and structures respectively. The value of a vector or structure can then be regarded as its pointer.

The concept of constancy must also be reviewed. The pointer itself may be constant or variable as may the elements or fields. The language must provide for each possibility.

7.1 Vectors

A vector provides a method of grouping together objects of the same type. Since S-algol does not allow undefined values all the initial values of the elements must be specified. The syntax is

```
BNF @< clause >of[c]< type >< bra >< clause.list >< ket >
TR  @{ int }of[c]< type>< bra >{ T },{ T },....{ T }< ket > => { *..T }
```

e.g.

```
@1 of int [ 1,2,3,4 ]
```

is a vector of integers, i.e *int, with lower bound 1 and values 1, 2, 3 and 4. The elements are variable.

Also

```
let abc := @1 of int [ 1,2,3,4 ]
```

introduces a variable "abc" of type *int and that initial value.

Multi dimensional vectors which are not necessarily rectangular can also be created.

e.g.

```
let Pascal = @1 of c*cint[
                          @1 of cint[ 1 ],
                          @1 of cint[ 1,2,1 ],
                          @1 of cint[ 1,3,3,1 ],
                          @1 of cint[ 1,4,6,4,1 ],
                          @1 of cint[ 1,5,10,10,5,1 ] ]
```

Pascal is of type **int. However it is constant as are all its elements. This is a fixed table.

This form of vector expression is sometimes very tedious to write especially for large rectangular vector with a common value. Therefore another form of vector expression is available.

BNF vector < bounds > of < clause >

where < bounds > ::= < clause >::< clause >[,< bounds >]

TR vector { int }::{ int },....,{ int }::{ int } of { T } => { *.....T }

e.g.

```
vector -1::3 of -2
```

produces a five element integer vector with all the elements variable and initialised to -2.

In order that the compiler can check that no constant will be updated it is necessary that objects of type *T and type *cT are made incompatible. In practice this is a small restriction.

7.2 Upb and Lwb

Since vectors may be passed around, it is often necessary to interrogate the vector to find its bounds. The functions upb and lwb are provided in S-algol for this purpose.

7.3 Structures

Objects of different types can be grouped together into a structure. To ensure strong typing in the language the structure class and the fields have to be given names. The fields may of course be constant or variable.

Before a structure can be created the shape of its structure class must be declared.

< structure.decl > ::= structure< identifier >[(< s.dec.list >)]

< s.dec.list > ::= [c]< type >< identifier.list >[;< s.dec.list >]

e.g.

```
structure person( cstring name ; bool sex ; int age,height )
```

declares a structure class, person, with 4 fields of type string, bool, int and int respectively. The string field is constant. It also declares the field names, name, sex, age and height. To create an object of this type requires an expression of this type

```
person( "Ronald Morrison",true,33,175 )
```

This expression is of type pntr.

7.4 Is and Isnt

The user may wish to check that a pointer is of a certain class. The binary operators is and isnt are provided.

```
BNF        < exp > is < identifier >
TR         { pntr } is { ( T1,.....Tn )-structure } => { bool }
```

If the pointer is of the correct class is gives the result true and isnt gives false.

7.5 Indexing

To obtain the elements of a vector or the fields of a structure indexing is used. For a vector, the index is an integer and for a structure the field name is used.

```
BNF            < name >(< clause >)
TR vector      { *T }({ int }) => { T }
TR structures  { pntr }({ T-field }) => { T ]
```

Before any indexing is performed the bounds of the vector are checked against the index for legality and in the case of the structure, the class of the structure is checked.

The comma notation may be used for vectors or structures when the elements or fields are themselves pointers. The indexing of vectors and structures may therefore be mixed.

```
BNF            < name >(< clause >)
TR vector      { *,.....*T }({ int },{ int },.....{ int }) => { T }
TR structure   { pntr }({ T1-field },....{ Tn-field }) => { Tn }
```

7.6 Equality of Pointers

Equality is defined on all data objects in S-algol. In the case of compound objects equality means the equality of the pointer. That is, for two vectors or structures to be equal they must be the same incarnation of the same object.

8. Clauses

The expression is a special type of clause which allows the operators in the language to be used to produce data objects. There are other statements in S-algol which allow the data objects to be manipulated and some which are there to control the flow of the program. These are now discussed.

8.1 Assignment Clause

BNF < name > ::= < clause >

TR { T } := { T } => { void }

e.g.

```
discrim := b * b - 4.0 * a * c
```

gives discrim the value of the expression on the right. Of course, the name must have been declared as a variable and not a constant. The clause allows the value to be altered.

8.2 If Clause

There are two forms

BNF if< clause >do< clause >

TR if{ bool }do{ void } => { void }

This is the single pronged version. If the condition after the if is true then the clause after the do is executed.

e.g.

```
if a < b do a := 3
```

The second version is

BNF if< clause >then< clause >else< clause >

TR if{ bool }then{ T }else{ T } => { T }

This allows a choice between two actions to be made. If the first clause is true the second clause is executed, otherwise the third clause is executed. Notice that the second and third clauses are of the same type and the result is of that type.

e.g.

```
if x = 0 then y := 1 else x := y - 1

let temp = if a < b then 1 else 5
```

8.3 Case Clause

The case clause is a generalisation of the if clause which allows the selection of one item from a number of possible ones. The syntax is

BNF case< clause >of< case.list >default:< clause >

< case.list > ::= < clause.list >:< clause >[;< case.list >]

< clause.list> ::= < clause >[,< clause >]*

TR

```
case { T1 } of
{ T1 },{ T1 },........{ T1 } : { T2 }
.
.
.
{ T1 },{ T1 },........{ T1 } : { T2 }
default : { T2 } => { T2 }
```

e.g.

```
case next.car.colour of

1,4     : "green"

2       : "blue"

3       : "red"

default : "any"
```

The value next.car.colour is compared in strict order, i.e left to right, top to bottom, with the expressions on the left hand side of the colon. When a match is found the clause on the right hand side is executed. Control is then transferred to the next clause after the case clause. If no match is found then the default clause is executed. The above case clause has result type string.

8.4 Repeat While Do Clause

There are three forms of this clause which allow loops to be set up with the test at the start, the end or the middle of the loop.

BNF
repeat< clause >while< clause >[do< clause >]
while< clause >do< clause >

TR
repeat{ void }while{ bool }[do{ void }] => { void }
while{ bool }do{ void } => { void }

In each of the three forms the loop is executed until the boolean clause is false. The while do version is used to perform a loop zero or many times whereas the repeat while is used for one or many times. The third form is a mixture of the two.

8.5 For Clause

The for clause is included in the language as a bit of syntactic sugar where it is known in advance how many times the loop will be executed.

BNF for< identifier >=< clause >to< clause >[by< clause >]do< clause >

TR for< identifier >={ int }to{ int }[by{ int }]do{ void } => { void }

The clauses are the initial value, the limit, the increment and the clause to be repeated respectively. The first three are of type int and are calculated only once at the start. If the increment is 1 then the increment clause may be omitted. The identifier or control constant is declared at the start of the void clause taking on the values defined by initial value, increment and limit.

e.g.

```
let factorial := 1 ; let n = 8
for i = 1 to n do fact := fact * i
```

9. Sequences and Scope Rules

```
< sequence > ::= < declaration >[;< sequence >]|
                 < clause >[;< sequence >]
                 < empty >
```

with type rules

```
< declaration > => { void }
{ void } ; { T } => { T }
< empty > => { void }
```

That is, a sequence of clauses is made up of any mixture, in any order, of declarations and clauses. The type of the sequence is the type of the last clause or declaration. If there is more than one object in a sequence then all but the last must be of type void.

9.1 Brackets

Brackets are used to make a sequence of clauses and declarations into a single clause. They are

```
begin                    or        { < sequence > }
    < sequence >
end
```

The {} method is there to allow a clause to be written clearly on one line.

e.g.

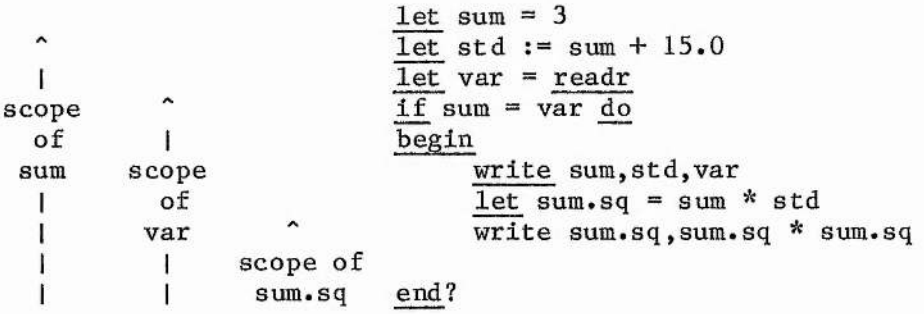
```
let i := 2
for j = 1 to 5 do { i := i * i ; write i }
```

However, if the clause is longer than one line the first alternative should be used for greater clarity.

9.2 Scope Rules

Any identifier that is declared has its scope limited to the following sequence. This means that the scope of an identifier starts immediately after the declaration and continues up to the next unmatched { or end.

e.g.



If the same identifier is declared in an inner sequence, then while the inner name is in scope the outer one is not.

10. Procedures

Procedures in S-algol constitute the abstractions over expressions, if they return a value, and clauses of type void if they do not. In accordance with the principle of correspondence any method of introducing a name in a declaration has an equivalent form as a parameter.

Thus, declarations of data objects giving a name an initial value is equivalent to assigning the actual parameter value to the formal parameter. Since this is the only type of declaration for data objects in the language, it is also the only parameter passing mode and is commonly known as "call by value".

Like declarations, the formal parameter representing data objects must have a name, a type and an indication of whether it is variable or constant. The name is an identifier, the type is the type name which is prepended with the letter c if the object is constant. A procedure which returns a value must also specify its return type.

Procedures, structure classes and fields may also be passed as parameters to complete the principle of correspondence. For type checking, the argument and result types of the procedures and the field types of structure classes must be given in full when they are passed as parameters. Whereas the constancy of a formal parameter of a procedure which is itself a formal parameter is immaterial it is not so with structure fields and to avoid the possibility of altering a constant, the constancy attribute of the fields must be the same in this case.

The syntax of the procedure declaration is

```
< proc.decl >     ::= procedure< identifier >[< T.spec >];< clause >
< T.spec >        ::= ([< param.list >][< arrow >< type >])
< param.list >    ::= < param.type >[;< param.list >]
< param.type >    ::= < typel >< identifier.list >|< structure.decl >|
                      < proc.type >< identifier.list >
< proc.type >     ::= ([< type2.list >][< arrow >< type >])
< type2.list >    ::= < typel >[,< type2.list >]|
                      < proc.type >[,< type2.list >]|
                      < structure.decl >[,< type2.list >]
< typel >         ::= [c]< type >
< arrow >         ::= ->
```

The procedure call has the following syntax

```
< identifier >[(< expression.list >)]
```

There must be a one-to-one correspondence between the actual and formal parameters and their types. The type of the above clause is the type of the procedure.

10.1 Forward Declarations

In S-algol all names must be declared before they can be used and a name comes into scope immediately after its declaration. This is awkward when recursive procedure definitions are involved. Therefore in the case of the procedure, the name comes into scope after the parameter list has been specified allowing procedures to call themselves.

A forward declaration is required for mutually recursive procedures. Its syntax is

forward < identifier >[< proc.type >]

This is merely an aid to the compiler. However, there is one rule with forward declarations that has nothing to do with the one pass nature of the compiler but with the decision to allow clauses and declarations to be freely mixed. To avoid the possibility of an uninitialised name being used by jumping over a declaration , a forward declaration and the actual declaration may only be separated by structure class declarations and other procedure declarations. In other words its use is restricted to mutual recursion situations.

11. Input and Output

The S-algol I/O system operates on files. A file is a collection of data. Two files, standard input and standard output which have the reserved identifiers s.i and s.o respectively, are of special interest. They represent in any implementation the standard input and output devices. In the language s.i and s.o are variables of type file. Other files may be created but this is implementation dependent.

11.1 Input

< read.clause > ::= < read.id >[(< file >)]

< read.id > ::= read|readi|readr|readb|reads|peek|
b.readi|b.readr|b.readb|b.reads

The read functions are in two forms. One regards the file as a series of characters and the other as binary digits. The two can be mixed freely but extra care should be taken when the binary form is used since no checking of the input is possible. The functions mean

peek	form the next character into a string. Do not advance the input stream.
read	form the next character into a string.
readi,b.readi	read an integer.
readr,b.readr	read a real.
readb,b.readb	read a boolean.
reads,b.reads	read a string.

In the character read, the characters are read until the literal of the required type is formed. In the binary case, only the correct number of bits from the internal representation of the object are read. The binary form is there to allow programs to pass information without having to code

it into characters.

The file descriptor may be omitted if the standard input is used.

e.g.

```
let a = readr
```

a is now a constant of type real with the value of the real literal just read. In the input stream, blanks, tabs and newlines before numbers are elided.

```
let c = reads( s.i )
```

reads a string literal, also from the standard input, and initialises the constant c to that value.

11.2 Output

BNF

```
< write.clause > ::= write< write.list >|[b.]output< clause >,< write.list >
< write.list >   ::= < clause >[:< clause >][,< write.list >]
```

TR

```
write { T1 }[:{ int }],{ T2 }[:{ int }],.....{ Tn }[:{ int }] => { void }
b.output { file },{ T1 }[:{ int }],.........{ Tn }[:{ int }] => { void }
```

write allows output to the standard output, s.o, only whereas output and b.output take a file descriptor as the first data object.

Example

```
write "I am O.K.",32160,3.4e-2,"Done"
```

will write that list of objects to the standard output and

```
write 13.2 : 16, 3 : 10, 152 : 3
```

will write 13.2, 3 and 152 to the standard output right justified in fields of width 16, 10 and 3 respectively.

Note that the field may be any integer expression allowing variable formats. If the field size is missing or the object cannot fit into it, then the object is written out in its exact size.

11.3 i.w, s.w and r.w

Types int and real have one other facility. The predefined integer variables

i.w	integer width initially 12
s.w	space width initially 2
r.w	real width initially 14

may be used to control output. s.w spaces are written out after any integer or real. i.w and r.w are used to set the field sizes for integers and real respectively, but may be overridden by the field size in the write clause. If the number fits neither then the exact size is written.

11.4 Eformat, Fformat and Gformat

These standard functions take a real number and return a string representing that number in roughly Fortran E, F or G format. Gforamat will return the string in either E or F format, whichever is appropriate.

The functions are

```
procedure fformat( real no ; int w,d -> string )
procedure eformat( real no ; int w,d -> string )
procedure gformat( real no -> string )
```

where

no	is the real to be converted
w	the number of places before the decimal point not including the sign
d	the number of places after the decimal point

Leading zeros are converted to blanks and the sign if negative is placed next to the number.

Thus

fformat(-31.6,3,2) would yield " -31.60"

and

eformat(32.3,1,2) would give " 3.23e+01"

gformat returns the real number in E or F format depending on its magnitude.

11.5 End of File

A test for end of file can be made using the standard function eof.

```
eof[(< file descriptor >)]
```

e.g.

```
while ~eof do write read
```

copies the input stream to the output stream until the end of the input stream is reached.

12. Standard Functions and Identifiers

S-algol defines a number of functions for the user. These reserved names may be applied as functions, but since they are really reserved words, they may not be passed as parameters. This restriction is easily overcome. For completeness they are all listed here.

```
procedure sqrt( real x -> real )
! the positive square root of x where x >= 0

procedure exp( real x -> real )
! e to the power x

procedure ln( real x -> real )
! the logarithm of x to the base e where x > 0

procedure sin( real x -> real )
! sine of x( radians )

procedure cos( real x -> real )
! cosine of x( radians )

procedure atan( real x -> real )
! arctangent of x ( radians ) where -π / 2 < atan( x ) < π / 2

procedure code( int n -> string )
! string of length 1 where s( 1|1 ) = character with
  numeric code abs( n rem 128 )

procedure decode( string s -> int )
! numeric code for s( 1|1 )

procedure upb( *T a -> int )
! upper bound of vector a

procedure lwb( *T a -> int )
! lower bound of vector a

procedure float( int n -> real )
! the value n as a real number

procedure truncate( real x -> int )
! the integer i such that |i| <= |x| < |i| + 1 where i * x >= 0

procedure abs( real x -> real )
! the absolute value of real number x

procedure abs( int n -> int )
! the absolute value of integer n
```

```
procedure length( string s -> int )
! the number of characters in the string s

procedure eformat( real n ; int w,d -> string )
! the string representing n in Fortran E format

procedure fformat( real n ; int w,d -> string )
! the string representing n in Fortran F format

procedure gformat( real n -> string )
! the string representing n in Fortran E or F format
```

plus the read functions listed in section 11.1.

12.1 Standard Identifiers

A number of standard identifiers exist in the language. They are

r.w	variable initially 14
s.w	variable initially 2
i.w	variable initially 12
s.i	variable set to the standard input
s.o	variable set to the standard output
maxint	constant, the maximum integer
epsilon	constant, the largest real e such that 1 + e = 1
pi	constant, pi
maxreal	constant, the largest real

13. References

1. Strachey, C. Fundamental Concepts in Programming Languages. Oxford University, 1967

2. Wirth, N. and Hoare, C.A.R. A contribution to the development of algol. Comm.ACM 9,6 (June 1966),413-431

3. Wirth, N. The programming language Pascal. Acta Informatica 1 (1971),35-63

4. Schuman, S.A. Towards modular programming in high level languages. Algol Bulletin 37 (1974),12-23

5. Legard, H.F. and Marcotty, M. A genealogy of control structures. Comm.ACM 18,11 (November 1975),629-639

6. Tennent, R.D. Language design methods based on semantic principles. Acta Informatica 8 (1977),97-112

7. Gunn, H.I.E. and Morrison, R. On the implementation of constants. Information Processing Letters 9,1 (July 1979),1-4

8. Lindsay, C.H. Modals. Algol Bulletin 37.4.3

9. Turner D.A.,Morrison R., Towards portable compilers. University of St.Andrews TR/76/5

Appendix A

S-algol Syntax

```
< program > ::= < sequence >?

< sequence > ::= < declaration >[;< sequence >]|
                 < clause >[;< sequence >]|
                 < empty >

< clause > ::= if< clause >do< clause >|
               if< clause >then< clause >else< clause >|
               repeat< clause >while< clause >[do< clause >]|
               while< clause >do< clause >|
               for< identifier >=< clause >to< clause >
                 [by< clause >]do< clause >|
               case< clause >of< case.list >default:< clause >|
               < name >:=< clause >|
               < write >|
               < expression >

< expression > ::= < exp1 >[or< exp1 >]*

< exp1 > ::= < exp2 >[and< exp2 >]*

< exp2 > ::= [~]< exp3 >[< relop >< exp3 >]

< exp3 > ::= < exp4 >[< addop >< exp4 >]*

< exp4 > ::= [< addop >]< exp5 >[< multop >< exp5 >]*

< exp5 > ::= < name >|
             < literal >|
             (< clause >)|
             {< sequence >}|
             begin< sequence >end|
             < name >(< clause >< bar > < clause >)|
             @< clause >of< typel >< bra >< clause.list >< ket >
             vector< bounds >of< clause >|

< name > ::= < identifier >|< expression >[(< clause.list >)]*

< clause.list > ::= < clause >[,< clause >]*

< case.list > ::= < clause.list >:< clause >;[< case.list >]

< bounds > ::= < clause >::< clause >[,< bounds >]

< bra > ::= [

< ket > ::= ]

< star > ::= *

< bar > ::= |
```

```
< addop > ::= +|-

< multop > ::= < star >|/|div|rem|++

< relop > ::= is|isnt|<|>|<=|>=|~=|=

< write > ::= write< write.list >|[b.]output< clause >,< write.list >|

< write.list > ::= < clause >[:< clause >][,< write.list >]

< declaration > ::= < let.decl >|
                    < structure.decl >|
                    < proc.decl >|
                    < forward >

< let.decl >       ::= let< identifier >< init.op >< clause >

< init.op >        ::= =|:=

< structure.decl > ::=  structure< identifier >(< field.list >)

< field.list >     ::= < typel >< identifier.list >[;< field.list >]

< proc.decl >    ::= procedure< identifier >[< T.spec >];< clause >

< T.spec >       ::= ([< param.list >][< arrow >< type >])

< param.list >   ::= < param.type >[;< param.list >]

< param.type >   ::= < typel >< identifier.list >|< structure.decl >|
                     < proc.type >< identifier.list >

< proc.type >    ::= ([< type2.list >][< arrow >< type >])

< type2.list >   ::= < typel >[,< type2.list >]|
                     < proc.type >[,< type2.list >]|
                     < structure.decl >[,< type2.list >]

< typel >        ::= [c]< type >

< arrow >        ::= ->

< forward > ::= forward< identifier >[< proc.type >]

< type > ::= int|real|bool|string|pntr|file|< star >< typel >

< identifier.list > ::= < identifier >[,< identifier >]*

< identifier > ::= < id >|< standard.id >

< id > ::= < letter >|< id >< letter >|
                 < id >< digit >|< id >.

< digit > ::= 0|1|2|3|4|5|6|7|8|9
```

```
< letter > ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|
               A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

< literal > ::= true|false|
                < u.int >[ .< u.int > ][ e< scale.factor > ]|"[< char >]*"

< char > ::= any ascii character

< u.int > ::= < digit >[ < digit > ]*

< scale.factor > ::= [ < addop > ]< u.int >

< standard.id > ::= code | decode | upb | lwb | float | eof
                    read | readi | readr | readb | reads | read.name |
                    b.readi | b.readr | b.readb | b.reads |
                    sin | cos | exp | ln | sqrt | atan | truncate |
                    abs | length | fformat | eformat | fiddle.r |
                    r.w | i.w | s.w | s.o | s.i |
                    maxint | maxreal | epsilon | pi
```

Appendix B

Type Matching Rules

```
< program >< eof > => { void }

< declaration > => { void }
{ void } ; { T } => { T }
< empty > => { void }

if { bool } do { void } => { void }
if { bool } then { T } else { T } => { T }
repeat { void } while { bool } [ do { void } ] => { void }
while { bool } do { void } => { void }
for < identifier > = { int } to { int }
                    [ by { int }] do { void } => { void }
case { T1 } of { T1 },{ T1 }....{ T1 } : { T2 }
               .
               .
               .
               default : { T2 } => { T2 }
{ T } := { T } => { void }
write { T1 },{ T2 },{ T3 }.......... => { void }
[b.]output { file },{ T1 },{ T2 }......... => { void }

{ bool } < or | and > { bool } => { bool }
~ { bool } => { bool }
{ int }< <|>|<=|>= >{ int } => { bool }
{ real }< <|>|<=|>= >{ real } => { bool }
{ string }< <|>|<=|>= >{ string } => { bool }
{ T } < =|~= > { T } => { bool }
{ int } < +|- > { int } => { int }
{ real } < +|- > { real } => { real }
< +|- > { int } => { int }
< +|- > { real } => { real }
{ int } < *|div|rem > { int } => { int }
{ real } < *|/ > { real } => { real }
{ pntr } < is|isnt > (.......)-structure => { bool }
{ string } ++ { string } => { string }
{ string }( { int } | { int } ) => { string }

( { T } ) => { T }
begin { T } end => { T }
< T-literal > => { T }

< T proc >[({ T1 },{ T2 },.......{ Tn })] => { T }

vector { int } :: { int },.......
       { int } :: { int } of { T } => { *......*T }
@{ int }of[c]< type >< bra >{ T },{ T },....{ T }< ket > => { *T }
< structure class >[({ T1 }, .......{ Tn } )] => { pntr }

{ *T }( { int } ) => { T }
{ pntr }( { T-field } ) => { T }
```

Appendix C

Unix Files

The following functions are available for use in the UNIX operating system. They are used to manipulate files and correspond to Section II of the Unix Programmer's Manual which should be consulted before use.

```
procedure open( string name ; int mode -> file )
! open the file named and return the file descriptor.

procedure close( file f )
! close the file f.

procedure flush( file f )
! clear out the internal buffer for file f. This should be done
  before closing the file after writing.

procedure create( string name ; int mode -> file )
! create the named file and return the file descriptor.

procedure seek( file f ; int offset,ptrname )
! see SEEK(II)
```

If the procedures create and open are unsuccessful, the value nullfile is returned. The reserved word nullfile is a file literal in Unix S-algol.

Appendix D

Program Layout

Semi-Colons

As a lexical rule in S-algol, a semi-colon may be omitted whenever it is used as a separator and it coincides with a newline. This, of course, allows many of the annoying semi-colons in a program to be left out.

However, to help the compiler deduce where the semi-colons should be, it is a rule that a line may not begin with an infix operator which is used as an infix operator.

e.g.

```
a *
b      is valid
```

but

```
a
* b        is not
```

This rule also applies to the invisible operator between a vector or pntr and its index list.

e.g.

```
let b = a( 1,2 )        is valid
```

but

```
let b = a
        ( 1,2 )
```

will be misinterpreted since vectors can be assigned

Comments

Comments may be placed in a program by using the symbol "!". Anything between the ! and the end of the line is regarded by the compiler as a comment.

e.g. a + b ! add a and b

Directives

There are certain compiler directives that the user may wish to invoke. The symbol "%" is used to denote a directive. They are

%list	print the program listing.
%nolist	do not print the program listing.
%title,< string literal >	take a new page and use the string as a heading for this and subsequent pages.
%lines,< integer literal >	Inform the compiler of the number of lines on each page of output paper.
%ul	underline the reserved words in the listing.
%noul	turn off the reserved word underlining.

Appendix E

Unix System

A number of commands are available in the Unix system to run S-algol. S-algol runs in two parts, a compilation followed by an execution. The compiler takes a source file, the name of which must end in .S, and produces the code in the file s.out. A listing of the program and error messages are given on the standard output. The compiler is invoked by the command Scom and it may have two options. The first option allows the user to specify that a cross reference listing is required. This will create the files X.DATA and X.DECL which are used later to produce the cross reference. The second options allows the user to produce a flow summary, i.e. a count of how many times each line of code is executed. This option produces the file listing.SS which is used later. Therefore to compile the source file Ron.S with the two options, the following command line is used

```
Scom Ron.S flsum xref
```

After the compilation the cross reference listing may be produced on the standard output by the command

```
Sxref
```

The program may be executed by the command

```
Sint
```

If a file other than s.out is to be executed, the Sint command will take the file name prepended by a minus sign as a parameter. Thus

```
Sint -ron
```

will execute the S-algol code in file ron.

If the program was compiled with the flow summary option, the file count.SS will be created on execution and may be used in conjunction with the file listing.SS by the command

```
Sflsum
```

to produce the flow summary on the standard output.

Finally, the Unix version of S-algol is extended to allow access to the Unix command options by providing a function which will return them as a vector of constant strings. It is defined by

```
procedure options( -> *cstring )
```

Appendix F

Sample Programs

```
                S-algol System          page    1

    3 --  !Towers of Hanoi
    4 --  procedure move( cstring a,b )
    5 --  write " ",a,"->",b,"'n"
    6 --
    7 --  procedure hanoi( cint n ; cstring a,b,c )
    8 --  if n > 0 do
    9 1-  begin
   10 --        hanoi( n-1,a,c,b )
   11 --        move( a,b )
   12 --        hanoi( n-1,c,b,a )
   13 -1 end
   14 --
   15 --  hanoi( 9,"a","b","c" )
   16 --  ?
**** Program Compiles ****
```

```
                S-algol System          page    1

    3 --  ! Sieve of Eratosthenes
    4 --  write "Input highest number :-  "
    5 --  let n = readi
    6 --  let p = vector 2 :: n of true
    7 --
    8 --  for i = 2 to truncate( sqrt( n ) ) do
    9 --  if p( i ) do
   10 --  for j = 2 * i to n by i do
   11 --  p( j ) := false
   12 --
   13 --  s.w := 2
   14 --  write "The primes less than ",n : 4," are :-'n"
   15 --  let fc := 0
   16 --  for i = 2 to n do
   17 --  if p( i ) do
   18 1-  begin
   19 --        write i : 5
   20 --        fc := fc + 1
   21 --        if fc = 8 do { write "'n" ; fc := 0 }
   22 -1 end
   23 --  ?
**** Program Compiles ****
```

```
                S-algol System          page    1

  3 --  let domino = vector 1::7,1::2 of 0
  4 --
  5 --  procedure swap( cint i,j )
  6 1-  begin
  7 --        let hold := domino( i,1 )
  8 --        domino( i,1 ) := domino( j,1 ) ; domino( j,1 ) := hold
  9 --        hold := domino( i,2 )
 10 --        domino( i,2 ) := domino( j,2 ) ; domino( j,2 ) := hold
 11 -1  end
 12 --
 13 --  procedure greater( cint i,j -> bool )
 14 1-  begin
 15 --        let sumi = domino( i,1 ) + domino( i,2 )
 16 --        let sumj = domino( j,1 ) + domino( j,2 )
 17 --        sumi > sumj or
 18 --        sumi = sumj and ( domino( i,1 ) = domino( i,2 ) or
 19 --                          domino( j,1 ) ~= domino( j,2 ) and
 20 --                          domino( i,1 ) > domino( j,1 ) )
 21 -1  end
 22 --
 23 --  !                   Main Program
 24 --
 25 --  write "Input the dominos 'n"
 26 --  for i = 1 to 7 do
 27 1-  begin
 28 --        let x := readi ; let y := readi
 29 --        if x < y do { let hold = x ; x := y ; y := hold }
 30 --        domino( i,1 ) := x ; domino( i,2 ) := y
 31 -1  end
 32 --
 33 --  for i = 1 to 6 do
 34 --  for j = i + 1 to 7 do if ~greater( i,j ) do swap( i,j )
 35 --
 36 --  write "Hand in preferred order is'n"
 37 --  s.w := 0
 38 --  for i = 1 to 7 do write domino( i,1 ) : 1,"|",domino( i,2 ) : 1," "
 39 --  ?
**** Program Compiles ****
```

```
              S-algol System          page    1

  3 --  let s := "" ; let n := 0 ; let not.done := true
  4 --
  5 --  procedure add.a.letter ; { s := s ++ "a" ; n := n + 1 }
  6 --
  7 --  procedure alter
  8 --  case s( n|1 ) of
  9 --  "a"     : s := s( 1|n-1 ) ++ "b"
 10 --  "b"     : s := s( 1|n-1 ) ++ "c"
 11 1-  default : { n := n - 1 ; if n = 0 then not.done := false
 12 -1                                   else alter }
 13 --
 14 --  procedure acceptable( -> bool )
 15 1-  begin
 16 --       let ok := true ; let p := 1
 17 --       while ok and p <= n div 2 do
 18 2-       begin
 19 --            ok := s( n - p + 1|p ) ~= s( n - 2 * p + 1|p )
 20 --            p := p + 1
 21 -2       end
 22 --       ok
 23 -1  end
 24 --
 25 --  !                  Main Program
 26 --
 27 --  write "Input string length "
 28 --  let lnth = readi ; let count := 0
 29 --  while not.done and n <= lnth do
 30 1-  begin
 31 --       if n = lnth then alter else add.a.letter
 32 --       while not.done and ~acceptable do alter
 33 2-       if n = lnth do { write "An acceptable string is ",s,"'n"
 34 -2                        count := count + 1 }
 35 -1  end
 36 --  write "Total number of strings of length ",lnth : 3,"is ",
 37 --        count : 4,"'n"
 38 --  ?
**** Program Compiles ****
```

```
                S-algol System          page    1

    3 --  write "Input three real coefficients "
    4 --  let a = readr ; let b = readr ; let c = readr
    5 --  let discrim = b * b - 4.0 * a * c ; let dem = 2.0 * a
    6 --
    7 --  case true of
    8 --  discrim < -epsilon : write "Imaginary Roots are ",-b / dem,
    9 --                               " + or - i * ",sqrt( -discrim )/dem,"'n"
   10 1-  discrim > epsilon : begin
   11 --                            let r1 = ( -b + sqrt( discrim ) )/dem
   12 --                            write "Real Roots are ",r1,
   13 --                                  " and ",c/( a * r1 ),"'n"
   14 -1                       end
   15 --  default     : write "Single Roots are ",-b/dem,"'n"
   16 --  ?
**** Program Compiles ****
```

```
                S-algol System          page    1

    3 --  procedure integral( ( real -> real ) F ; creal a,b -> real )
    4 --  ( b - a ) * F( ( a + b ) / 2.0 )
    5 --
    6 --  procedure G( creal z -> real )
    7 1-  begin
    8 --        procedure ep( creal y -> real ) ; exp( z * y )
    9 --
   10 --        integral( ep,3.0,4.0 )
   11 -1  end
   12 --
   13 --  write fformat( integral( G,1.0,2.0 ),4,2 ),"'n"
   14 --  ?
**** Program Compiles ****
```

Appendix II

Measurement Tools

********** Execution Flow Summary **********

```
 1 --      0   %
 2 --      0   %
 3 --      1   let s := "" ; let n := 0 ; let not.done := true
 4 --      0
 5 --    706   procedure add.a.letter ; { s := s ++ "a" ; n := n + 1 }
 6 --      0
 7 --      0   procedure alter
 8 --   2118   case s( n|1 ) of
 9 --   2118   "a"     : s := s( 1|n-1 ) ++ "b"
10 --   1412   "b"     : s := s( 1|n-1 ) ++ "c"
11 1-    706   default : { n := n - 1 ; if n = 0 then not.done := false
12 -1    705                                    else alter }
13 --      0
14 --      0   procedure acceptable( -> bool )
15 1-   2118   begin
16 --   2118        let ok := true ; let p := 1
17 --   8487        while ok and p <= n div 2 do
18 2-   6369        begin
19 --   6369             ok := s( n - p + 1|p ) ~= s( n - 2 * p + 1|p )
20 --   6369             p := p + 1
21 -2      0        end
22 --   2118        ok
23 -1      0   end
24 --      0
25 --      0   !                  Main Program
26 --      0
27 --      1   let lnth = 12 ; let count := 0
28 --    971   while not.done and n <= lnth do
29 1-    970   begin
30 --    970        if n = lnth then alter else add.a.letter
31 --   2119        while not.done and ~acceptable do alter
32 2-    970        if n = lnth do { write "An acceptable string is ",s,"'n"
33 -2    264                         count := count + 1 }
34 -1      0   end
35 --      1   write "Total number of strings of length ",lnth : 3,"is ",
36 --      1         count : 4,"'n"
37 --      0   ?
```

Dynamic flow analysis of S-code instructions

Nine bit operations		Six bit operations		Four bit operations	
ge.op	0	local	58964	jump	12283
gt.op	0	localaddr	13002	apply.op	4942
le.op	8308	plocal	529	jumpf	14223
lt.op	0	plocaladdr	0	for.test	0
eq.op	2646	dlocal	0	jumpff	11577
neq.op	6369	mst.local	4237	newline	47981
not.op	2118	global	25724	jumptt	0
no.op	0	globaladdr	1413	for.step	0
plus.op	20077	pglobal	16974		
times.op	6369	pglobaladdr	2118		
minus.op	14856	dglobal	0		
divide.op	0	mst.global	705		
div.op	7338	make.vector	0		
rem.op	0	retract	2119		
neg.op	0	form.structure	0		
no.op	0	store.sf	3		
subs.op	0				
subsaddr	0				
cjump	3530				
ass.op	16533				
return	4942				
forward.op	0				
suba.op	0				
subaaddr	0				
ldc.pntr	1				
ldc.bool	2120				
ldc.si	43056				
ldc.real	0				
ldc.string	6179				
ldc.li	0				
no.op	0				
no.op	0				
load	0				
loadaddr	0				
pload	0				
ploadaddr	0				
dload	0				
mst.load	0				
no.op	0				
enter	4943				
float.op	0				
is.op	0				
isnt.op	0				
upb.op	0				
lwb.op	0				
finish.op	1				
reverse.real	0				
erase.op	971				
read.op	0				
write.op	797				
iliffe.op	0				
no.op	0				

arith.op	0
io.op	0
b.read.op	0
b.write.op	0
code.op	0
decode.op	0
substr.op	16268
concat.op	2118
length.op	0
options.op	0

The operations in order are

local	58964
newline	47981
ldc.si	43056
global	25724
plus.op	20077
pglobal	16974
ass.op	16533
substr.op	16268
minus.op	14856
jumpf	14223
localaddr	13002
jump	12283
jumpff	11577
le.op	8308
div.op	7338
times.op	6369
neq.op	6369
ldc.string	6179
enter	4943
return	4942
apply.op	4942
mst.local	4237
cjump	3530
eq.op	2646
ldc.bool	2120
retract	2119
concat.op	2118
not.op	2118
pglobaladdr	2118
globaladdr	1413
erase.op	971
write.op	797
mst.global	705
plocal	529
store.sf	3
finish.op	1
ldc.pntr	1

The total number of operations = 386334

Static analysis of S-code

Nine bit operations		Six bit operations		Four bit operations	
ge.op	0	local	19	jump	9
gt.op	0	localaddr	3	apply.op	5
le.op	2	plocal	3	jumpf	6
lt.op	0	plocaladdr	0	for.test	0
eq.op	3	dlocal	0	jumpff	3
neq.op	1	mst.local	4	newline	23
not.op	1	global	9	jumptt	0
no.op	0	globaladdr	3	for.step	0
plus.op	5	pglobal	6		
times.op	1	pglobaladdr	3		
minus.op	5	dglobal	0		
divide.op	0	mst.global	1		
div.op	1	make.vector	0		
rem.op	0	retract	2		
neg.op	0	form.structure	0		
no.op	0	store.sf	3		
subs.op	0				
subsaddr	0				
cjump	2				
ass.op	9				
return	3				
forward.op	0				
suba.op	0				
subaaddr	0				
ldc.pntr	1				
ldc.bool	3				
ldc.si	26				
ldc.real	0				
ldc.string	10				
ldc.li	0				
no.op	0				
no.op	0				
load	0				
loadaddr	0				
pload	0				
ploadaddr	0				
dload	0				
mst.load	0				
no.op	0				
enter	4				
float.op	0				
is.op	0				
isnt.op	0				
upb.op	0				
lwb.op	0				
finish.op	1				
reverse.real	0				
erase.op	3				
read.op	0				
write.op	8				
iliffe.op	0				
no.op	0				

arith.op	0
io.op	0
b.read.op	0
b.write.op	0
code.op	0
decode.op	0
substr.op	5
concat.op	3
length.op	0
options	0

Number of string elements = 70

Number of bits used for operation codes is = 1393

Huffman coding total bits = 925

Instruction	Frequency	Level	Binary Coding
finish.op	1	8	11010010
ldc.pntr	1	8	11010011
div.op	1	8	11010000
times.op	1	8	11010001
not.op	1	9	111101110
neq.op	1	9	111101111
mst.global	1	8	11110110
cjump	2	7	1100110
le.op	2	7	1100111
retract	2	7	1111010
concat.op	3	6	010100
erase.op	3	6	010101
ldc.bool	3	6	100010
return	3	6	100011
eq.op	3	6	100000
store.sf	3	6	100001
pglobaladdr	3	6	100110
globaladdr	3	6	100111
plocal	3	6	100100
localaddr	3	6	100101
jumpff	3	6	110010
enter	4	6	110101
mst.local	4	6	111100
substr.op	5	5	00110
minus.op	5	5	00111
plus.op	5	5	00100
apply.op	5	5	00101
pglobal	6	5	01011
jumpf	6	5	11000
write.op	8	5	11011
ass.op	9	5	11111
global	9	5	11100
jump	9	5	11101
ldc.string	10	4	0100
local	19	3	000
newline	23	3	011
ldc.si	26	3	101

Appendix III

The Abstract Machine Code

The S-algol abstract machine code, S-code, is designed to fit exactly the needs of the S-algol language. Appendix IV describes the code generated for each syntactic construct and Appendix V the format of each instruction in the PDP11 implementation. Here the individual instructions are described. They fall naturally into groups.

Jumps

jump(l)	unconditional jump to address l
jumpf(l)	jump to l if the top stack element is false. Remove the top element of the stack
jumptt(l)	jump to l if the top element is true. Otherwise remove the top stack element
jumpff(l)	jump to l if the top stack element is false. Otherwise remove the top stack element
cjump(t,l)	t is the type and determines which stack to use. If the top two stack elements are equal, remove both and jump to l. Otherwise remove only the top stack element. Be careful on equality of strings
fortest.op(l)	The control constant, increment and limit are the top three elements of the stack. If the increment is negative and the control constant is less than the limit or the increment is positive and the control constant is greater than the limit then remove them from the stack and jump to l
forstep.op(l)	Update the control constant by adding the increment. Then jump to l

Stack Load Instructions

These instructions are used to load any data item that is in scope, on to the top of the stack. The data items may be in the local, global or intermediate environments and a separate instruction exists for each form. Different instructions are also used for the separate stacks. The local and global forms of the instruction have a parameter which is the displacement of the item from the stack frame base. The intermediate form of the instruction requires the number of times to chain down the static chain as well as the displacement. Only one form of each type is described.

local(n),global(n),load(r,n)	load on the main stack
plocal(n),pglobal(n),pload(r,n)	load on the pointer stack
localaddr(n),globaladdr(n),loadaddr(r,n)	load address on the main stack
plocaladdr(n),pglobaladdr(n),ploadaddr(r,n)	load the address of the pointer stack item on the main stack
dlocal(n),dglobal(n),dload(r,n)	load double length item main stack

Relational Operations

The relational operations act on the data types int, real and string. The top two elements of the stack are compared and removed. The boolean result true or false is left on the main stack. The instructions have a parameter to indicate the type. Care should again be taken in the equality of strings. Equality is defined on all the data objects in the language.

ge.op(t)	greater than or equal to
gt.op(t)	greater than
le.op(t)	less than or equal
lt.op(t)	less than
eq.op(t)	equal to
neq.op(t)	not equal to

Arithmetic Operators

These instructions operate on the data types real and integer. The top two elements of the stack are replaced by the result except for negate and float which use only the top element.

plus.op(t)	add
times.op(t)	multiply
minus.op(t)	subtract
divide.op	divide real
div.op	divide int leaving quotient
rem.op	divide int leaving remainder
neg.op	negate
float.op	coerce the int to a real

Procedure Entry and Exit

The code to execute a procedure is surrounded by the two instructions enter and return.

enter(p,q) Check that the main stack has p cells left and the pointer stack has q cells

return(t) This is executed on procedure exit. The SF and PSF registers are updated. The SF register is first set to the current dynamic link and then PSF is set to the pointer stack link of the uncovered stack frame. The stack tops must be altered to remove the MSCW and any local items. The new stack tops are the current pointer stack link for the pointer stack and the position of the MSCW for the main stack. If the type of the procedure is not void, the result must be copied to the new stack top

The code sequence to call a procedure is

```
mst.load
evaluate the parameters
apply.op
```

The code for the evaluation of the parameters is the same as for any expression. The mark stack and load instruction, loads the procedure closure and leaves space on the stack for the rest of the MSCW. The apply

instruction fills in the MSCW with the dynamic link and the pointer stack link. These are calculated from the new stack top minus the space already allocated in these frames. Since procedure names follow the same scope rules as any other name, there are three forms of the instruction.

mst.local(n),mst.global(n),mst.load(r,n)	load the procedure closure from the stack and leave space for the rest of the MSCW
apply.op(m,n)	fill in the dynamic link (SP - m), the pointer stack link (psp - n) and the return address. Update SF and PSF and jump to the address pointed at by SF

There are two further instructions involved with procedures.

forward.op	leave space for the procedure closure on the stack
store.sf(n)	place the procedure closure on the stack. If the address n is not the top of the stack, it is a forward declared procedure and n is its stack address

Vector and Structure Creation Instructions

These instructions take information off the stack and create heap objects. These objects are then initialised and the pointer to them left on the top of the pointer stack.

make.vector(t,m,n)	t indicates the type of the objects in the vector and therefore on which stack they reside. m points to the position of the lower bound on the main stack. The difference between PSP and n or SP and m depending on which stack is in use, gives the number of vector elements. The instruction creates a vector and fills in the elements. The stack pointers are then reduced to m and n with the pointer to the vector being placed on the pointer stack
iliffe.op(t,n)	t indicates the base type. n pairs of bounds are on the main stack. However, the top of one of the stacks will contain the initial value. The instruction creates an iliffe vector of the shape indicated by the bound pairs and the value of the initial expression is copied into the elements of the last dimension. The expression value and the bound pairs are removed from the stack and the pointer to the vector is placed on the pointer stack

form.structure(n) The expressions which initialise the structure fields have been evaluated on the appropriates stacks. n points to the trademark on the main stack. A structure of the correct size is made up and the fields filled in. To do this the structure table is referred to, to give the number of pointer fields. After removing the fields from the stacks the pointer to the structure is placed on the pointer stack

Vector and Structure Accessing Instructions

These instructions are generated by the compiler to index a vector or a structure. The index of the vector must be checked against the bounds before the indexing is done. Similarly the structure class (trademark) of a structure must be checked.

suba.op(t) The vector index is on the top of the main stack and the vector pointer on the pointer stack. These are used to check that the index is legal and then to find the required value. They are removed from the stack and replaced by the value, on the stack indicated by the result type t

subs.op(t) The structure pointer is on the top of the pointer stack. The main stack contains the trademark and the field address. The trademark is checked against the structure trademark and if it is the same the field address is added to the pointer to yield the absolute field address. The trademark, field address and the structure pointer are replaced on the stack by the result whose type is t

subaaddr(t) This is the same as suba.op except that the address of the item is calculated and placed on the main stack

subsaddr This is the same as subs.op except that the address of the item is calculated and placed on the main stack

lwb remove the pointer to the vector from the pointer stack and place its lower bound on the main stack

upb remove the pointer to the vector from the pointer stack and place its upper bound on the main stack

is.op The trademark is on the main stack and is compared with the trademark of the structure pointed at by the top element of the pointer stack. Remove both and place the boolean result of the comparison on the main stack

isnt.op This is the same as is.op except it has the opposite test

Load Literal Instructions

These are used to load the value of a literal on to the stack. The literal usually follows the instruction in the code stream and so the CP register has to be updated accordingly.

ll.pntr	load the nil pointer on to the pointer stack
ll.bool(n)	load the boolean value n (true or false) on to the main stack
ll.sint(n)	load the value of a short integer (-64 to 63) on to the main stack
ll.real(n)	load the real on to the main stack
ll.string(s)	load the string address on to the pointer stack
ll.lint	load a long integer, 16 bits, on to the main stack

String Operations

These are used to perform the string operations in S-algol.

code.op	The integer n at the top of the stack is removed and a string of length 1 with character |n| rem 128 placed on the top of the pointer stack.
decode.op	The string at the top of the pointer stack is removed and the value of its first character placed on the main stack as an integer
length.op	The string at the top of the pointer stack is removed and its length placed on the main stack
concat.op	remove the two strings from the top of the pointer stack and replace them with a new string which is the concatenation of them
substr.op	A new string is created from the one at the top of the pointer stack and replaces it. It is formed by using the length at the top of the main stack and the starting position at the second top. After checking that these are legal they are removed

Arithmetic Functions

There are a number of arithmetic functions supported by the language which take a real value off the main stack and replace it by the result. They are

sin, cos, exp, ln, atan, truncate, abs

Truncate takes a real and leaves an integer, and abs may map from int to int or real to real. These functions are defined in the language reference manual.

Input and Output

The s-code machine supports both binary and character I/O streams. For every character stream instruction there is an equivalent binary stream one. Only the character ones are given here.

read.op(n)	the stream descriptor is on the top of the pointer stack. This is removed and the value read is placed on the appropriate stack. n indicates which read function to use. They are
read	read a character and form it into a string
reads	read a string
readi	read an integer
readr	read a real
readb	read a boolean
read.name	read an S-algol identifier and form it into a string
peek	same as read but do not advance the input stream
write.op(t)	The field width is on the top of the main stack and the item to be written out either under it or on the pointer stack. The stream descriptor is under all this on the pointer stack. The field width and the item are removed from the stack. The type t of the write may be
write.int	write an integer
write.real	write a real
write.bool	write a boolean
write.string	write a string

If the field width is not specified then i.w and r.w come into use for int and real. s.w spaces are always written after integers or reals for character streams.

Format Operations

There are three format operations in S-code which take a real number and return a string representing that number in Fortran E, F or G format.

eformat.op,fformat.op	This takes the real number and the number of places before and after the decimal point and returns a string
gformat	Takes the real number and returns a string

Miscellaneous

reverse.stack(t)	Swap the top two elements of the stack indicated by type t
erase.op(t)	remove an element from the stack indicated by t
finish.op	stop the program execution
not.op	perform a not on the boolean at the top of the stack
ass.op	assign the value at the top of the stack to the address on the main stack and remove them
retract(t,n,m)	retract the main stack to n and the pointer stack to m. If t is not void move the value at the old stack top to the new stack top

Appendix IV

S-code Generated by the S-algol Compiler

A summary of the S-code generated by the S-algol compiler for each syntactic construct is given here. In the description E, in the source code represents an expression and E, in the code represents the S-code for that expression. Sometimes the expressions are of type void. A description of the instructions themselves is given in Appendix III.

Source	S-code
~E	E not.op
+E	E
-E	E neg.op
unary.function(E)	E unary.function.op
write E1:E1′,.....En:En′	s.o E1 E1′ write.op..... ...En En′ write.op erase.op

Write operates for reals, ints, bools and strings.

output E0,E1:E1′....En:En′	E0 E1 E1′ write.op..... .....En En′ write.op erase.op

similarly for binary output.

read	s.i read.op
read(E)	E read.op

similarly for peek, read.name, reads, readi, readb, eof.

E1 := E2	E1* E2 ass.op

where E1* is an L-value which may generate a load address.

E1(E2)*	E1 E2 subsaddr or subaaddr
E1 or E2	E1 jumptt(1) E2 1:
E1 and E2	E1 jumpff(1) E2 1:
E1 <binary.op> E2	E1 E2 binary.op
(E)	E
E1(E2|E3)	E1 E2 E3 substr.op
E1(E2)	E1 E2 subs.op or suba.op
E(E1,.....En)	mst.load E E1....En apply.op
@E of T[E1,....En]	E E1.....En make.vector
E(E1,.....En)	E E1.....En formvec.op
vector E1::E1′,...En::En′ of E	E1 E1′...En En′ E iliffe.op
if E1 do E2	E1 jumpf(1) E2 1:
if E1 then E2 else E3	E1 jumpf(1) E2 jump(m) 1: E3 m:

```
repeat E1 while E2                  1: E1 E2 not.op jumpf(1)
repeat E1 while E2 do E3            1: E1 E2 jumpf(m) E3 jump(1) m:
while E1 do E2                      1: E1 jumppf(m) E2 jump(1) m:
for I=E1 to E2 by E3 do E4          E1 E2 E3 1: fortest.op(m) E4
                                    forstep.op(1) m:

let I = E                           E
let I := E                          E
procedure I ; E                     enter E return
structure I                         11.int

<literal>                           11.literal dependent on type
<identifier>                        load.stack
```

A load.stack instruction may be one of load, local, global, pload, plocal, pglobal, dload, dlocal or dglobal.

The unary functions are code, decode, length, upb, lwb, float, sin, cos, exp, ln, sqrt, atan, truncate, abs, eformat, gformat and fformat.

The binary operations are eq.op, neq.op, lt.op, le.op, gt.op, ge.op, plus.op, times.op, minus.op, div.op, rem.op, divide.op, is.op, isnt.op and concat.op.

```
case E0 of                                  E0
E11,E12,......E1n : E10        E11 cjump(11) E12 cjump(11).....E1n cjump(11)
                               jump(M1) 11 : E0 jump(xit)
E21,E22,......E2n : E20     M1:E21 ..........
  .
  .
  .
  .
  .
default : Ek+1 0             Mk:Ek+1 0
                            xit:
```

Appendix V

The PDP11 S-code Operation Codes

4 Bit Operations

bit 15	1	to identify these
bits 12-14		operation code
bits 0-11		code address

The operation codes are

0	jump	1	apply.op
2	jumpf	3	for.test
4	jumpff	5	newline
6	jumptt	7	for.step

Newline uses bit 0-11 for the line number.
Apply.op uses bits 0-4 for the pstack retraction and bits 5-11 for the main stack retraction values. The pstack number is half the correct value.

6 Bit Operations

bits 14-15	01	to identify these
bits 10-13		operation codes
bits 0-9		stack address

The operation codes are

0	local	1	localaddr
2	plocal	3	plocaladdr
4	dlocal	5	mst.local
6	global	7	globaladdr
8	pglobal	9	pglobaladdr
10	dglobal	11	mst.global
12	make.vector	13	retract
14	form.structure	15	store.closure

Retract uses bits 0-9 as the address on the main stack to retract to and another word in the same format indicating the block type (bits 10-15) and the pstack retraction (bits 0-9). Make.vector is the same.
Store.closure takes a second word giving the procedure address.

9 Bit Operations

bits 14-15	00	to identify these
bits 7-13		operation codes
bits 0-6		usually the type

The operations are

0	ge.op	1	gt.op
2	le.op	3	lt.op
4	eq.op	5	neq.op
6	not.op		

Not.op does not require a parameter

8	plus.op	9	times.op
10	minus.op	11	divide.op
12	div.op	13	rem.op
14	neg.op		
16	subs.op	17	subsaddr
18	cjump	19	ass.op
20	return	21	forward.op
22	suba.op	23	subaaddr

Forward.op does not require a parameter and cjump uses a second word for the jump address.

24	ll.pointer	25	ll.bool
26	ll.sint	27	ll.real
28	ll.string	29	ll.lint

These are the load literals. The pointer is used for the empty string and the UNIX nullfile has only one possible value. The boolean has two values and the short integer values between -64 and 63. The long integer is followed by a word holding the integer and a real is followed by two words. The string instruction is followed by the string in its heap format.

32	load	33	loadaddr
34	pload	35	ploadaddr
36	dload	37	mst.load
38		39	enter

Enter uses bits 0-2 and bits 10-15 of the second word as the maximum main stack address and bits 0-9 of the second word for the maximum pstack address.
The other instructions use bits 0-6 as the reverse lexicographic level and a second word as the stack displacement.

40 float.op	41 is.op
42 isnt.op	43 upb.op
44 lwb.op	45 finish.op
46 reverse.stack	47 erase.op

Is.op, isnt.op and finish.op have no parameter.
Reverse.stack codes bits 0-6 as 0 for main stack, 1 for pstack.
Float.op uses bits 0-6 to indicate which stack element 0-top, 1-2nd. top.

48 read.op	49 write.op
50 iliffe.op	
52 arith.func	53 is.op
54 b.read	55 b.write

Iliffe.op uses a second word to indicate the number of bound pairs.

The codings for read and b.read are

0	readi	b.readi	1 readr	b.readr
2	readb	b.readb	3 reads	b.reads
4	peek	b.peek	5 read	b.read
6	read.name	b.read.name		

and for write and b.write

0 int	1 real
2 bool	
4 string	5 fformat
6 eformat	7 gformat

The arithmetic functions are coded thus

1 sin	2 cos
3 exp	4 ln
5 sqrt	6 atan
7 truncate	8 integer abs
9 real abs	10 fiddle.r

56 code.op	57 decode.op
58 substring.op	59 concat.op
60 length.op	61 options.op

These do not have a parameter.

In general the type codings for the 9 bit operations are

0 int	1 real
2 bool	3 file
4 pntr	5 void
6 string	7 vector